UNCLASSIFIED

A Comprehensive Context for Mobile-Code Deployment

Final Project Report for UC Irvine

Michael Franz

Grant Agreement No. N00014-01-1-0854
1st May 2001 – 30th September 2004

UNIVERSITY OF CALIFORNIA, IRVINE

Notice

Effort sponsored by the the Office of Naval Research under agreement No. N00014-01-1-0854. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 2005	**2. REPORT TYPE** Final report	**3. DATES COVERED** *(From - To)* 01 May 2001 – 30 Sept 2004

4. TITLE AND SUBTITLE A Comprehensive Context for Mobile-Code Deployment	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER N00014-01-1-0854
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Franz, Michael	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California Bren School of Information & Computer Science Irvine, CA 92697-7600	**8. PERFORMING ORGANIZATION REPORT NUMBER** 23071
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Ballston Centre Tower One 800 North Quincy Street Arlington, VA 22217-5660	**10. SPONSOR/MONITOR'S ACRONYM(S)** ONR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approve for Public Release, distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Given the acknowledged importance of mobile code, current distribution models are surprisingly primitive. For example, Java's model assumes that the constituent parts that make up a mobile program will all be downloaded to a single location, and then verified, linked, possibly dynamically compiled, and finally executed at that very location.

This research project made three important contributions: First, it demonstrated that it can be beneficial to perform verification, dynamic compilation, and execution at different physical locations. A prototype was built that performs code verification and just-in-time compilation at a "code generating router" inside the network itself. If the end-points of the network are resource-limited devices such as wirelessly-connected personal digital assistants (PDA's), off-loading dynamic code generation to the stationary network can result in substantial benefits. As a second contribution, the project identified a novel attack on mobile code systems, based on the complexity of the code verification algorithm itself. The third contribution is a new mobile-code verification algorithm that not only lacks this vulnerability, but that is also more efficient.

15. SUBJECT TERMS
mobile code distribution, dynamic code generation, code verification, verification complexity

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Michael Franz
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	SAR	69	**19b. TELEPHONE NUMBER** *(include area code)* (949) 824-7427

Abstract

Given the acknowledged importance of mobile code, current mobile-code distribution models are surprisingly primitive. For example, Java's model assumes that the constituent parts that make up a mobile program will all be downloaded to a single location, and then verified, linked, possibly dynamically compiled, and finally executed at that very location.

This research project has made three important contributions: First, it demonstrated that it can be beneficial to perform verification, dynamic compilation, and execution at different physical locations. A prototype was built that performs code verification and just-in-time compilation at a "code generating router" inside the network itself. If the end-points of the network are resource-limited devices such as wirelessly-connected personal digital assistants (PDAs), off-loading dynamic code generation to the stationary network can result in substantial benefits.

As a second contribution, the project identified a novel attack on mobile code systems, based on the complexity of the code verification algorithm itself. The third contribution is a new mobile-code verification algorithm that not only lacks this vulnerability, but that is also more efficient.

Contents

List of Figures

Acknowledgments

The author would like to thank all team members for their research contributions to this project, and would like to recognize especially Deepak Chandra, Andreas Gal, Vivek Haldar, Christian Probst, Vasanth Venkatachalam, and Lei Wang.

Summary

Given the acknowledged importance of mobile code, current mobile-code distribution models are surprisingly primitive. For example, Java's model assumes that the constituent parts that make up a mobile program will all be downloaded to a single location, and then verified, linked, possibly dynamically compiled, and finally executed at that very location.

This research project has made three important contributions: First, it demonstrated that it can be beneficial to perform verification, dynamic compilation, and execution at different physical locations. A prototype was built that performs code verification and just-in-time compilation at a "code generating router" inside the network itself. If the end-points of the network are resource-limited devices such as wirelessly-connected personal digital assistants (PDAs) or high-end mobile phones, off-loading dynamic code generation to the stationary network can result in substantial benefits.

As a second contribution, the project identified a novel attack on mobile code systems, based on the run-time complexity inherent in the code verification algorithm. The new attack is not based on any error in the implementation but exploits the underlying worst-case behavior of a published algorithm. Open-source implementations are particularly susceptible to this kind of algorithmic attack.

The third contribution is a new verification algorithm for Java that verifies bytecode via Static Single Assignment (SSA) form construction. The resulting SSA representation can immediately be used for optimization and code generation. The prototype implementation requires less time to transform Java bytecode into SSA form and verify it than it takes Sun's original Java verifier to merely confirm the validity of the bytecode, with the added benefit that SSA is available "for free" to later compilation stages.

1 Introduction

Mobile code is an important enabling technology with a huge potential, but the underlying software distribution models are surprisingly primitive. While major research investments have been made into dynamic compilation and related efforts to improve the *run-time performance* of programs distributed in this manner, much less has been invested in understanding and improving the *stages of the mobile-code pipeline* that bring the code to this point in the first place.

The predominant model by far, which for example underlies the distribution of Java "applets" over the Internet, identifies dynamically linkable parts of mobile programs by URL strings. The model further assumes that the constituent parts that make up a mobile program will be downloaded to a single location, and then verified, linked, possibly dynamically compiled, and finally executed at that very location. This model is unnecessarily simplistic, as it precludes many useful alternative deployment strategies, particularly in the realm of *network-connected embedded devices*.

The performance of the code distribution pipeline is particularly critical for such resource-limited embedded devices (e.g., wirelessly-connected personal digital assistants or high-end mobile phones). In order for such a device to run Java code, it needs to contain either a Java interpreter or a just-in-time compiler. Both options are costly: an interpreter needs to run at a high clock frequency to overcome the overhead of interpretation, while a just-in-time compiler consumes extra memory and processor resources for the compilation step.

Code Generating Routers: Our project has found a particularly interesting solution to this fundamental problem by integrating the just-in-time compiler into the network itself, in what we call a "code-generating router". The code-generating router translates mobile code into the native instruction set of the handheld device while it is passing through. For example, in a mobile telephony system, such a component could be co-located with the base station. A full description of this approach follows in Section 2.

Complexity-Based Attacks on Mobile Code: A second focus point of our research concerned itself with code verification. We predict that we will soon witness denial-of-service attacks on mobile-code systems that will be based on algorithmic complexity. For example, the worst-case performance of Java Bytecode Verification rises quadratically with program length. By sending a legal, but difficult-to-verify program to a server virtual machine, we can keep that server occupied for an inordinate amount of time, effectively making it unavailable for useful work. In our experiments, a 4kByte JAR file containing Java code with worst-case verification complexity required more than 15 minutes of verification time on a high-end workstation. Our result puts into question the premise of open-source software, since it is knowledge of the underlying algorithm that is exploited in the attack, rather than a particular implementation defect. This is described in more detail in Section 3.

SSA-Based Java Bytecode Verification: Java bytecode verification has other problems. If the verification task determines that the bytecode is indeed safe, then this code is forwarded in its original form (as received from the code producer) to the JVM's execution component, which may be an interpreter or a just-in-time compiler. Beyond the result denoting whether or not verification was successful, all other information computed by the verifier is discarded and is not passed onwards. In many cases, this results in a duplication of work when a just-in-time compiler subsequently performs a very similar data-flow analysis all over again. In the course of this project, we found an alternative verification mechanism that avoids such duplication of work. It is described in Section 4.

2 Code Generating Routers

With the ongoing expansion of wireless coverage, formerly autonomous devices such as Personal Digital Assistants (PDAs) and mobile sensors (MSs) have evolved into nodes of large distributed systems. Their increased connectivity and flexibility has unleashed an enormous potential for the execution of mobile code. Astonishingly, at the same time there has been no change to the role of the *network*: The execution model of mobile code is the same on small devices as it is on powerful workstations. Current approaches for PDAs and MSs rely on *node-local* just-in-time compilation or, even worse, on pure interpretation by virtual machines. In a resource-constrained setting, these approaches often produce unsatisfactory results.

We propose a paradigm shift toward a *network-centric* mobile code execution model. Instead of the traditional *device-centric* approach that embeds a just-in-time compiler component into every mobile device, we perform mobile code optimization and compilation effort *within the networking infrastructure*. In this approach, the networking infrastructure serves as a transparent proxy. It intercepts mobile code sent from a server to a mobile device and forwards optimized native code for immediate execution instead (Figure 1).

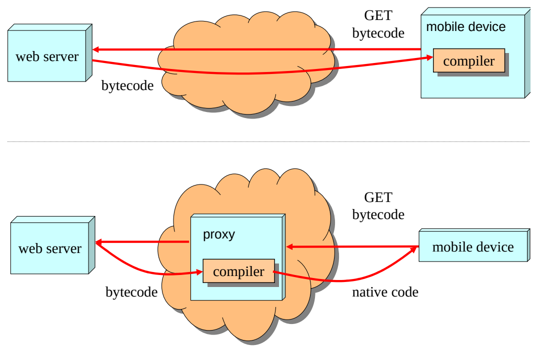


Figure 1: Traditional Mobile Code Execution vs. Network-based Compilation

This architecture turns the execution model from a decentralized, device oriented one, into a centralized, network oriented one, thereby saving resources on the mobile device in terms of processor time, memory and battery consumption. This is achieved without changing the current client-server architecture for mobile code deployment. This is a particularly important property since it provides for backward compatibility with existing mobile code solutions.

In the following, we describe the current scenario of mobile code execution (Section 2.1). This scenario is dominated by *replicated compilation*, where each and every terminal node is able to compile code. Network-centric compilation is a better approach (Section 2.2). The prototype implementation of the presented framework is described in Section 2.3. Section 2.4 describes related work of server based mobile code compilation. We conclude this part of our report with a summary and an outlook to future work (Section 2.5).

2.1 Replicated Compilation

In the traditional mobile code setting, a workstation-class client requests an application from a remote server, i.e. web-server, and the network delivers the mobile code without performing any modification. Once the mobile code has been delivered to the workstation it is either interpreted by a virtual machine (VM) or is translated to native machine code by a just-in-time compiler and then executed directly by the host CPU. Pure interpretation using a virtual machine is in many cases too slow even on a powerful workstation-class computer. In the mobile device scenario, interpreted execution is especially harmful since it requires high clock frequencies, which induce a higher energy consumption.

Mobile code compiled just-in-time into native code executes near the speed of code specifically compiled for the target CPU. However, the compilation process itself is resource-intensive in terms of memory consumption and CPU time. All components needed for the mobile code execution reside on each target machine, including the actual virtual machine, the just-in-time compiler and the runtime libraries. In a network of such machines, each machine is independently capable and responsible for compiling and optimizing mobile code during execution. This can be summarized as *replicated compilation* at every terminal node.

On an average workstation-class computer the resource overhead of replicated compilation is irrelevant. For example, the Java 2 Platform requires roughly 32MB of RAM for the execution of a simple program while a modern workstation is usually equipped with well over 256MB of RAM. CPU time is also usually no issue since workstation-class machines are often equipped with fast microprocessors clocked at Gigahertz speeds.

Small mobile devices such as PDAs, mobile sensors, and mobile phones, however, are equipped with far fewer resources in terms of CPU power and memory. Therefore, a more elaborated approach is needed to allow for the efficient execution of mobile code on such platforms. This is even more urgent for mobile sensors, where the available resources are even more constrained.

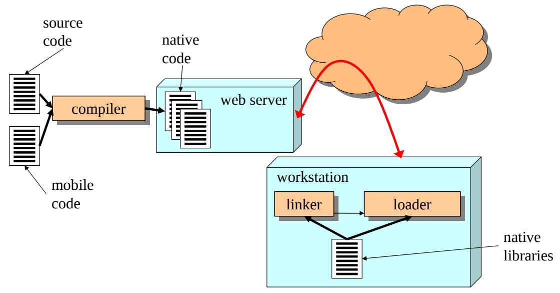


Figure 2: Execution of *Native* Mobile Code on a Workstation

A naive solution is to store *native* instead of *mobile* code on the server (Figure 2). Instead of transporting the portable mobile code to the target device, the mobile code is translated into machine-specific native code for the targeted architecture before it is put on the server. On the mobile device's side, the virtual machine and the just-in-time compiler are replaced by the standard linker and loader of the device's operating system. Additionally, the device's runtime libraries need to be replaced with native libraries. While this approach allows to reduce the runtime overhead by removing the virtual machine and the just-in-time compiler, the libraries still need to be present. This approach has been pursued by Microsoft before the advent of mobile code formats in their ActiveX framework.

One drawback of this approach is that the server has to store a version of each program for every target architecture that possibly could request a program from it. Another disadvantage lies in the nature of mobile code vs. machine code. Mobile code is transported in a type-safe manner, which allows to verify easily that certain security policies are not violated by the program. Performing the same verification on native machine code is much harder. Therefore, native code usually is secured by code signing when transported over unsecure networks. While type-safety is a *code-inherent* property, code signing relies on the safety of secret keys and the strength of signing algorithms.

Additionally, in this solution the overall role of the network with respect to the delivered code is still exactly the same as before. It *enables* communication between mobile devices and servers, but *remains inactive*. As can be seen in Figure 3, the networking infrastructure merely serves as a transportation medium.

Taking into account the readily available computing power of many network

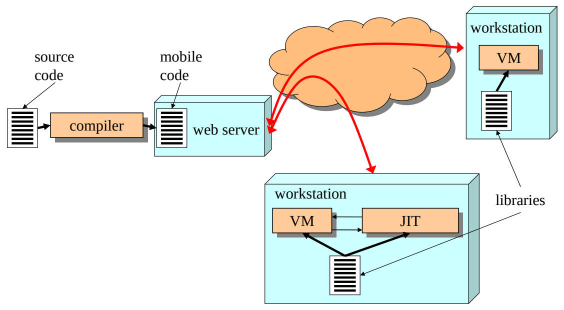


Figure 3: Execution of Mobile Code on a Workstation

nodes, we propose an alternative approach by enabling the networking infrastructure to compile mobile code, store compilation results and to deliver native code directly to the target devices.

2.2 Network-Centric Compilation

While recently there has been a lot of work on *active networks*, e.g. [74, 77], these approaches mostly think in terms of reprogrammable networking infrastructure, and mobile agents that perform this reprogramming. We extend this notion by modeling the network as an *enabling part* of the execution model for mobile code.

To off-load parts of the mobile code execution framework onto the network infrastructure, the compilation of mobile code into native code is delegated to a *code generating router* (CGR), which is also responsible for linking, and loading the native code into the memory of the target machine. Just as in the previous replicated compilation scenario, the mobile device requests an application from a remote server. The networking infrastructure, now playing an *active* role compared to its passive behavior in the previous section, intercepts this request at a router (Figure 4).

The active router then passes on the request to the remote server, initiating the transmission of the mobile code, but redirects the response stream to its internal compilation server. From the remote server's perspective this process is entirely transparent. This is an important property of our architecture as it allows to integrate our model with existing web services.

The compilation server inside the code generating router translates the mobile code to machine-code which is specifically geared toward the target device which issued the initial request. In contrast to a pure native-code approach such as ActiveX, native code is only exchanged over trusted communication links between the router and the target device (using, for example, encrypted wireless LAN).

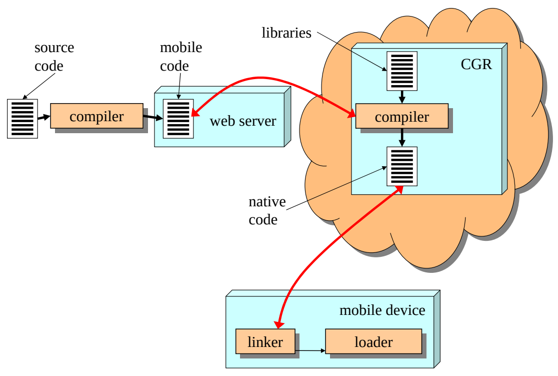


Figure 4: Network-Enabled Execution of Mobile Code on a Resource-Constrained Device

Remote servers still store and serve programs in verifiable mobile code format. When the target device requests a program from a remote host, the local networking infrastructure translates it on-the-fly to native machine code suitable for the target CPU. To eliminate the risks of injection of non-verifiable native code from foreign sources, the target device accepts native code input only from the local networking infrastructure. Therefore, the code generating router would usually be placed behind a firewall, securing the network between it and the ultimate host.

By relieving the target device from code generation and instead having the networking infrastructure taking over this task, we have effectively transformed the current *device-centric* mobile code execution framework into a *network-centric* model which performs the resource intensive compilation process where CPU time, memory, and battery power are aplenty and power consumption is much less critical than in the case of mobile devices.

Having the network take over functionality otherwise provided by code consumers or producers, the networking infrastructure automatically gains insight into the code executing on its nodes. Thus, i.e., the network is able to identify already compiled applications and parts and can reuse cached results instead of recompiling them. By storing the *intermediate representation* instead of the *generated code*, this benefit even holds if the same pieces of code are required for different architectures. This can be compared to web proxies, which store the *source* of web pages instead of the actual rendered output for a certain browser.

The network also relieves its client terminal nodes from storing libraries. Since the compilation to native code happens at routers in the network, neither the libraries nor the compilers need to be stored on the devices that actually execute

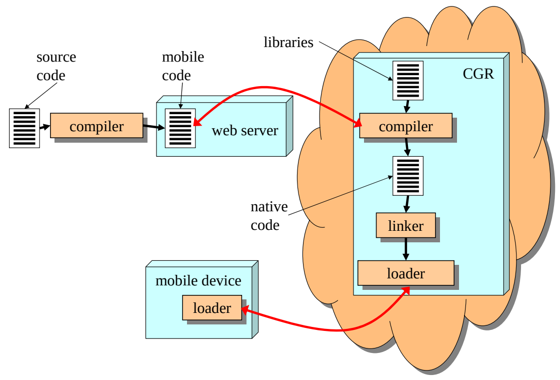


Figure 5: Network-Enabled Execution of Prelocated Mobile Code on a Resource-Constrained Device

the code. Thus, these devices are independent of version changes in either. Instead, the network delivers for each mobile-code request a native program compiled with the current version of all libraries.

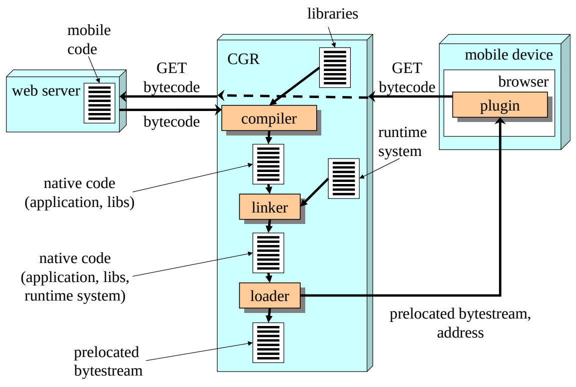


Figure 6: Communication in the Network-Centric Approach

Figure 5 shows how the resident footprint on the mobile device can be further reduced by delegating the *linking* process to the networking infrastructure as well. By doing so, the networking infrastructure gains complete control over the placement of code in the memory of each participating node. In order to perform the *linking* and even parts of the *loading* process, all it needs to know is the base address to which the resulting code shall be loaded. The active network generates

benchmark	#classes	bytecode [bytes]	specialized code [bytes]	opt. [bytes]	exec. time [s]	
					Jeode	CGR
HelloWorld	53	64,827	80,376	30,959	1.64	0.12
J2ME Test	53	65,878	82,096	37,287	57.15	3.03
method test	56	68,640	89,192	41,527	53.74	58.60
loop	53	64,623	80,456	33,307	1.88	0.09

Figure 7: Benchmark Results for Prototype Implementation

code, links it with the runtime system and performs *partial loading*, which we also call pre-relocation or *prelocation*. The resulting code image is sent to the target device, where the rest of the loading process is performed.

If a mobile device roams within the active network, it can be handed over between code generating routers. After the mobile device announces its new location, the nearest CGR will contact the previous CGR to download all relevant state information [26]. If the mobile devices leaves the active network entirely, it will not be able to download any more code from the CGR, but the already compiled code continues to work as expected.

2.3 Research Prototype

In this section we report about our prototype implementation of the networking-infrastructure based mobile code architecture. One of the main design goals of our prototype system was to re-use existing components and technologies where appropriate.

Due to its maturity and availability, we choose to use the Java language [35] and virtual machine [48] to be at the core of our implementation instead of designing our own mobile code representation. Besides the Java reference implementation offered by Sun Microsystems, a number of commercial and open source implementations of the Java virtual machine exist. Looking at the history and development of these projects, the implementation of the Java virtual machine itself is a fairly manageable task compared to the re-implementation or customization of the Java libraries which are mandatory to execute any program written in Java. To make matters worse, the Java API is frequently updated and changed by Sun Microsystems, making it very difficult for third party developers to keep their own Java libraries up to date. To prevent this problem, one of the fundamental design decisions for our prototype implementation was to stay compatible to the reference implementation of the Java API offered by Sun Microsystems.

As the target device we are using a Sharp Zaurus PDA with a 206 MHz Intel

StrongARM CPU and 16MB of RAM and 16MB of ROM. The functionality of a code generating router is performed by a standard workstation PC running our ProxyVM framework [76]. Our prototype implementation seamlessly integrates with the web browser that comes pre-installed on the Sharp Zaurus. All requests from the web browser to fetch Java bytecode (in form of Java class files) from remote web servers are intercepted by the network. The code generating router then reads the Java code from the web server, translates it to native code suitable for the Intel StrongARM CPU and relays the executable binary code back to the PDA.

The code generating router is not only responsible for verifying and compiling the Java mobile code to native code. After each Java class file has been compiled to native code, the networking infrastructure also connects the native code pieces into an executable program (*linking*) as well as adjusts all address references to the final memory location (RAM) of the code once it has been transfered to the PDA (*prelocation*). The final step of *loading* the native code into the memory of the PDA is also performed under the control of the network. The PDA receives specific instructions what code to write to what memory location. Thus, only minimal logic is required on the target device.

All that's necessary on the PDA to enable the execution of Java applets inside the web browser is a small browser plugin that follows the code loading instructions issued by the network and then transfers control to the newly loaded code. In contrast to existing solutions, no library code is stored on the PDA. Instead, only actually referenced parts of the library are transfered to the PDA. This is especially important, since the standard libraries come with a huge number of methods that are not callable at runtime.

The standard Java libraries are offered by Sun in a number of different flavors, including an edition for small embedded devices. Existing Java solutions for the Sharp Zaurus use a minimal Java API subset to save storage space on the PDA. In our architecture, the Java libraries are stored in the code generating router instead, enabling the execution of programs using the full Java API on small embedded devices[1].

Figure 7 and Figure 8 give results for compilation and execution with our prototype implementation. The benchmarks are `HelloWorld`, which is a treat in other programming languages, but in the Java world means compiling almost the complete Java API. The second benchmark, `J2ME Test`, is part of a small J2ME test suite [42]. It performs 100 iterations of arithmetic computations. `JGF Method` is based on the method invocation part of the JavaGrande benchmark [10].

[1]As a reference, the compressed full Java API (JDK 1.4.1) is approximately 25MB in size. This is more than four times the total amount of available RAM on the PDA.

It repeatedly executes calls to methods that are in either the same or another class as the calling method. Finally, the `loop` benchmark executes a loop a number of times. The results in Figure 7 have been measured with an upper bound of $10,000,000$.

All runtime measurements have been performed on a *Sharp Zaurus SL-5500*, using the native code generated by our framework and the original bytecode executed by the Jeode virtual machine [40]. Figure 7 gives two kinds of information for every benchmark — (1) the different code sizes, and (2) execution times for the mobile and the specialized code. The sizes are given for the mobile code (including the API classes) and the the generated, target specialized code, on the one hand for a normal compiler run and on the other hand for an optimizing run. Size is of importance for code that is going to be transported over the network, since smaller numbers obviously imply shorter transmission times. The code generated without optimizations averages approximately 126% of the size of the mobile code. With enabled optimizations the code size shrinks to 55%, allowing for a much faster code deployment. The small footprint is mainly due to applying aggressive optimizations, including Rapid Type Analysis [7, 75] which allows the compiler to eliminate all methods that are never going to be called at run time.

The second number for each benchmark is the actual execution time of the code. While the numbers for *HelloWorld* and *loop* are merely given for reasons of completeness, they share an interesting property that we are going to come back to shortly — since the computation performed in this benchmarks is negligible, the dramatic overhead of Jeode must actually stem from the initialization of the run-time system. Thus, beside the short transmission times resulting from the reduced code size of our target specialized code, the resulting application in our solution is going to start up much faster than a bytecode-based alternative. This is going to be even improved in the future by ordering the methods in the resulting byte-stream in the order that they might be called at execution time.

The benchmark execution times show the superiority of our solution in terms of the fast startup and the aggressive optimizations performed by our compiler. The only exception is the JavaGrande based benchmark which is 5% faster when executed in the Jeode environment than when the binary generated by our CGR is executed. Inspecting the generated code reveals that there are two main reasons for this — we conservatively ensure that classes are properly initialized, thus enforcing too many checks at run-time. Additionally, access to class and instance variables is still sub-optimal in our generated code. Since this benchmark in our setting invokes in the order of $8 * 10^8$ methods and counts in instance variables, these numbers are going to improve with an improved field access in the next generation of the infrastructure.

Figure 8 shows the execution time (y-axis) for the *loop* benchmark with

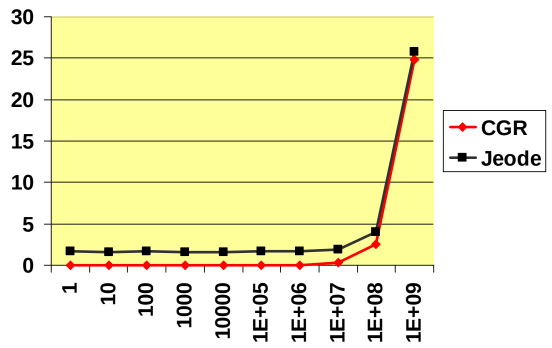


Figure 8: Execution Time of Loop Benchmark

different arguments (x-axis) measured with Jeode and our CGR. As can be seen, the difference between the two curves stays more or less constant, indicating that the code generated by our solution has a much shorter startup time, thus allowing for a faster availability of downloaded applications.

2.4 Related Work

Besides the Standard Edition [69] of the Java 2 Platform [35, 48], which is not directly suited for small embedded devices, a number of Java implementations exist that are directed at similar devices as our proposed architecture.

Sun Microsystems itself has published at least two Java virtual machines that are specifically designed for small embedded devices and PDAs. The Spotless System [11] is a small virtual machine for the Palm organizer using a minimal Java API subset. It contains no just-in-time compiler, but speed was no design goal. Similar to the Spotless System is the K Virtual Machine [52], also from Sun. While "K" stands for kilobyte, in a useful configuration the KVM needs closer to one megabyte of RAM than merely kilobytes. Like the Spotless System, KVM has no just-in-time compiler and is rather slow. Both systems have in common that they require all libraries to reside on the target system.

More recently, Sun has announced a project called KJIT which aims at producing a just-in-time compiler for the KVM system. A similar just-in-time compilation solution for small embedded systems is already offered by Insignia. The Jeode virtual machine [40] uses profiling and on-demand code generation to speed up the execution of Java code. The Sharp Zaurus PDA ships with a version of the Jeode virtual machine for the StrongARM CPU. While Jeode is much smaller than the Standard Edition of Java, it still occupies very significant amounts of ROM and RAM compared with the few kilobytes of code our architecture requires on the PDA.

Even smaller Java implementations such as JavaCard [73] succeed in further reducing the virtual machine overhead by dropping essential virtual machine features like class loading, reflection, floating point, long integer, etc. Our approach differs from these subset virtual machines by trying to reconcile reduced resource consumption on the mobile device and execution performance. By using Rapid Type Analysis, we are able to execute Java programs written for the standard Java API 1.4 on a device which has not even enough memory to hold just the API bytecode itself in compressed form, not to mention the actual program or the virtual machine code.

Similar to our architecture, JCOD [23] is using a network connection to offload the compilation of Java mobile code to a compilation server. Our approach differs from JCOD as we try to shift the responsibility for mobile code compilation and preparation for execution more aggressively into the networking infrastructure. In our architecture, all that remains of the Java virtual machine on the target device is a small plugin stub. Compilation, linking, prelocation, and loading are all performed by the code generating router in a manner entirely transparent to the mobile device. Finally, in contrast to JCOD, we compile all code to native code and never interpret any portions of the program on the target device and thus do not require any interpreter components on the target.

As our prototype implementation, JPure [8] translates mobile code in its entirety into machine code before execution. While being originally designed for small embedded controllers, the JPure project endeavored to scale Java down to devices typical to that domain. The main reasons were the inefficient code generation performed by the GNU Java Compiler (GCJ) and the non-availability of small Java API subsets at that time. In contrast to JPure, our prototype implementation is able to mimic the dynamic class loading semantics of Java and thus does not require to partially rewrite the Java libraries. This is a practical benefit as we do not have to track the frequent changes Sun Microsystems is doing on the Java API.

2.5 Section Summary

Bringing mobile code technology to small embedded devices has been an ongoing struggle since the inception of Java. With the advent of handheld devices such as PDAs, mobile sensors, and next generation mobile phones, the necessity of such a move has been further emphasized. We achieve off-loading computational intensive mobile code compilation and optimization tasks to the networking infrastructure. A central role in this novel architecture is an active networking component, a code generating router, which transparently intercepts mobile code requests and compiles code on-the-fly to a format directly (natively) executable by the mobile code requester. Already in this early stage of development, our prototype most of the time outperforms or at least equals existing stand-alone Java solutions for the chosen Sharp Zaurus platform in both major criteria: memory consumption and execution performance.

In contrast to advancements at the operating system and application programming level, just-in-time compilation (or code compilation in general) is currently still mostly considered a local task. Our architecture and the prototype implementation thereof have demonstrated that replicated compilation is not the favorable architectural choice in an ubiquitous computing environment. Already today, networking components such as routers and switches contain some of the most advanced and highest performance CPUs in existence. We believe that this trend will continue and will ultimately allow the networking infrastructure to contribute significantly to long power autonomy of portable digital assistants or networked portable devices in general.

Our plans for future work can roughly be divided into three areas. First, we want to further improve our mobile code compilation and optimization framework to overcome certain existing performance bottlenecks we pointed out in the previous sections. Our second major goal is to make more efficient use of information gained from past compilation runs in the CGR. Specifically, we plan to cache intermediate results of the compilation and optimization phase to speed up future compilation requests of the same mobile code fragment. Finally, we are exploring how profiling information gathered by the target device can be fed back to the CGR to perform runtime code optimization. Our current approach is based solely on static code analysis and we expect to be able to further increase execution performance by taking dynamic profile information into account.

3 Complexity-Based Attacks on Mobile Code

Safe mobile code is a major accomplishment. The two leading standards, the Java Virtual Machine [48] and the "Dot-Net" Common Language Runtime [50], provide target-machine independence in a code distribution format that can be verified by the code recipient prior to execution. Safety in such systems is based on a combination of (1) code verification ahead of execution and (2) runtime monitoring and resource control during execution. The research emphasis in this area so far has been on the *correctness* of the safety mechanism and its implementation, and numerous vulnerabilities have been discovered and removed.

In this section, we contend that mere correctness of the safety enforcement mechanism is not sufficient to defend the host computer against mobile-code based attacks. Instead, there needs to be a dual focus on the (worst-case-)*performance* of the verifier and just-in-time compiler. Otherwise, as we will show, formally correct safe-mobile-code systems are vulnerable to a new class of denial-of-service (DoS) attack that exploits the *complexity characteristics* of the underlying verification and code-generation algorithms to render the system unavailable for useful work. Since the attack is located ahead of the point at which run-time resource control sets in, and since it attacks the very mechanism upon which safety is founded, conventional defenses cannot fully quell this threat.

For example, the Java virtual machine bytecode verification algorithm exhibits quadratic worst-case execution complexity. We have been able to construct relatively small mobile programs in the Java JAR archive format that require hours of verification on high-end workstations. The programs in question are perfectly legal Java virtual machine code, perform no malicious action on the host, and will eventually be verified as being safe. However, the process of verification itself is so costly as to effectively constitute a denial-of-service attack.

Current mobile code systems treat verification and just-in-time code generation as atomic operations. As far as we know, there is not a single existing Java Virtual Machine in which verification or just-in-time code generation can be interrupted by the user—other than by killing the whole Virtual Machine. And for the increasingly important *server-side virtual machines*, human intervention is not even an option.

The problem with the kind of attack that we describe in this section is that the programs in question are not "illegal" in the sense that traditional safety mechanisms would defend against. In fact, there might be completely reasonable *valid and useful* programs with verification complexities similar to our attack programs. Hence, one cannot simply deploy a traditional monitor that would abort verification when a certain time limit is exceeded—unless one wants to also accept the random rejection of potentially important non-malicious programs. The risk of rejecting certain useful programs might be particularly unacceptable for an unattended server virtual machine.

Hence, the problem is similar to that of defending against low-intensity viruses or worms that cause damage while "flying under the radar" without ever tipping off an intrusion-detection system. Unlike such a low-intensity virus or worm, however, the attack in our case consumes all of the cycles of the host.

Unfortunately, attacks based on algorithmic complexity affect not just the verifier, but the complete code path on the client. For example, an adversary that knows the target virtual machine's register allocation algorithm might be able to maliciously craft a valid mobile code program containing a particularly difficult to solve graph coloring puzzle.

An interesting point to note is that “open source” systems might be more vulnerable to this kind of attack than systems that provide “security by obscurity”. Advocates of “open source” development have long argued that their systems are safer because the code is audited by hundreds of people and implementation defects are hence more easily spotted and removed. Our attack, however, does not depend on any implementation defect, but only on the underlying *algorithm*, which is publicly exposed in open-source development. Hence, the open-source process simultaneously increases the vulnerability to attacks such as ours while making it impossible to quickly react to an exposed vulnerability by changing the underlying algorithm.

The remainder of this part of our report is structured as follows: The next section gives a short introduction to mobile code verification, followed by a description of the current state of Java security (Section 3.2). This chapter also discusses successful earlier attacks and the related countermeasures that have been taken. Section 3.5 presents a whole class of new attacks based on complexity and defines the problem by identifying vulnerabilities in mobile-code execution frameworks that have so far been overlooked. Here, we contend that a paradigm shift is necessary towards making mobile code systems aware of complexity-based attacks and hardening them against those attacks. In Section 3.11, we argue that the solution may very well lie in choosing algorithms based on their *worst-case*, rather than *average-case* behavior. Currently, virtual machines and just-in-time compilers are fine tuned for high performance in the average case, with the hope that the worst case will rarely occur—we assert that this approach is simply too dangerous. We end this section with a brief summary (Section 3.12).

3.1 Mobile Code Verification

Static mobile code verification was introduced by Gosling and Yellin [78, 48] as an alternative to the *dynamic* checking of type safety properties at runtime through dynamic execution monitors [15]. Using dynamic runtime checks can cause a significant runtime overhead at execution time. The basic ingredient of every Java virtual machine bytecode verifier is an abstract interpreter for Java Virtual Machine Language instructions. The stacks and virtual registers of this abstract interpreter store *types*, rather than *values*. Similarily, the instructions of the abstract interpreter operate at the type-level only and do not perform any actual calculations.

Leroy [47] lists the minimal conditions for bytecode to be accepted by the verifier:

- *Type correctness*. Bytecode instructions are typed and must receive arguments of appropriate types.

- *No stack overflow or underflow*. A method must never pop a value from the empty stack or push a value onto the largest stack specified for that method.
- *Code containment*. The program counter must always stay within the code limits of the currently active method and must always point to the beginning of a valid instruction.
- *Local variable initialization*. No variable may be loaded that has not been initialized first.
- *Object initialization*. Whenever an object of a class C is created, one of the class' constructors must be called.

Except for code containment, all of these conditions require tracking the types of values as they are pushed onto and popped from the virtual Java stack and written to and read from local variables. The Java specification provides an outline of a data-flow algorithm that can be used for this purpose. A simplified description of this algorithm is shown in Figure 9.

All Java Virtual Machine implementations that we are aware of, including Sun's own CVM [71] and HotSpot virtual machines [41], use slight variations of this algorithm to perform bytecode verification. The verifier algorithm is performed separately for every method in a Java program. For each method, it iterates over all instructions of that method until no more operand type changes are observed. For each instruction i, the verifier checks whether the abstract data associated with i has changed. If so, it checks whether the current abstract local variable and stack content allow the execution of i and computes the new local variable and stack content. Finally, this new abstract state is propagated to all successors of i.

3.2 The Current State of Java Security

The Java Virtual Machine [48] and its security architecture [78, 49] have been under intense scrutiny since their release. Over the years, several errors on different levels have been unveiled. Figure 10 gives an overview of the structure of the Java bytecode-execution framework and references to reported flaws and shortcomings in either the implementation or the architecture. The whole security concept of Java collapses at the moment at which just *one* of the components is compromised. Below, we will give a brief overview over some of these attacks. It is noteworthy that all of these attacks are based on *implementation* errors rather than *conceptual* flaws. In contrast, the attack presented in this section, is not as its enabling property is the byte-code verification algorithm itself and not a specific implementation thereof.

```
 1: $todo \leftarrow true$
 2: while $todo = true$ do
 3:   $todo \leftarrow false$
 4:   for all $i$ in all instructions of a method do
 5:     if $i$ was changed then
 6:       $todo \leftarrow true$
 7:       check whether stack and local variable types
          match definition of $i$
 8:       calculate new state after $i$
 9:       for all $s$ in all successor instructions of $i$ do
10:         if current state for $s \neq$ new state derived from $i$ then
11:           assume state after $i$ as new entry state for $s$
12:           mark $s$ as changed
13:         end if
14:       end for
15:     end if
16:   end for
17: end while
```

Figure 9: The Standard Verification Algorithm Found in Sun Microsystem's JVM Implementations.

Beside the work described in this section, there has also been intensive research on how to actually secure the transport and the execution of mobile-code programs. Approaches include enhanced transport formats [2, 53, 16], host based intrusion-detection systems [3, 22], auditing systems [64], stack inspection [25], and extensions to the actual execution unit [6, 19, 37]. All of these approaches either try to rescue what already has been lost—the system is no longer dependable due to probable implementation faults—or try to replace the current verifier scheme with mechanisms that are hopefully easier to implement and prove correct.

3.3 Implementation-Based Attacks

In attacking a mobile-code execution framework, the verifier is one of the obvious targets. The verifier has the obligation to prohibit any execution of unsafe code, where *unsafe* is defined with respect to the criteria defined in the respective framework. Due to its importance, verification is executed as a highly prioritized task that can not be interrupted by the user without shutting down the whole virtual machine. There are two possible scenarios for an attack on the bytecode verifier.

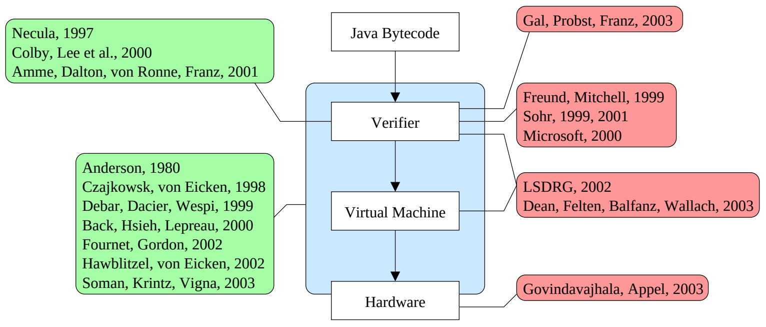

Java Bytecode
Verifier
Virtual Machine
Hardware
Necula, 1997
Colby, Lee et al., 2000
Amme, Dalton, von Ronne, Franz, 2001
Anderson, 1980
Czajkowsk, von Eicken, 1998
Debar, Dacier, Wespi, 1999
Back, Hsieh, Lepreau, 2000
Fournet, Gordon, 2002
Hawblitzel, von Eicken, 2002
Soman, Krintz, Vigna, 2003
Gal, Probst, Franz, 2003
Freund, Mitchell, 1999
Sohr, 1999, 2001
Microsoft, 2000
LSDRG, 2002
Dean, Felten, Balfanz, Wallach, 2003
Govindavajhala, Appel, 2003

Figure 10: High-Level Structure of a Bytecode-Execution Framework. The call-outs on the right-hand side refer to successful attacks and implementation flaws found in the JVM, the call-outs on the left-hand side to attempts to enhance the security and reliability of the virtual machine.

Obviously, for attacking a virtual machine it is of importance to bring a hostile applet through the verification process. Since the verifier has the obligation to prohibit exactly this, this task requires profound inside knowledge and usually profits from implementation errors.

Sohr [62] presents an example for the exploit of faulty implementations. In this attack, the verifier *overlooks* certain code sequences that are passed on for execution *without having been verified*. This, in turn, allows to construct methods that use the unverified code to return objects of one type that by the type system are believed to be of another type. In later work, Sohr [63] reports on a similar problem in a version of Microsoft's bytecode verifier [51], based on incorrect handling of values of local variables in the verifier.

Freund et al. [29] exploit another flaw in a certain implementation of the verifier, which is based on the handling of subroutines and uninitialized objects in the verifier. By creating several uninitialized objects of a class and only initializing one of them, the virtual machine can be made to invoke methods on uninitialized objects. This fault is enabled by incorrect handling of initialization of newly created objects in the verifier.

Hopwood [39] presents another exploit demonstrating that certain versions of the verifier could be made to load classes with absolute class names instead of relative ones. Thus, a class could first be uploaded to a client without verification and afterwards be dynamically loaded in the virtual machine. However, since it would be loaded from the local host, there would not be any verification. This trust in local files is applied to avoid repeated re-verification of locally installed classes such as the pre-installed system classes of the APIs of mobile-code execution frameworks.

The actual hardware running the virtual machine can also be attacked, e.g. by raising the probability of a bit error [36]. In contrast to the exploits described so far, this is an area of attack that can not be handled by means of the virtual machine itself, but by taking hardware measures to minimize the probability of an uncorrectable memory error.

3.4 Class Loader-Based Attacks

The first attack that included the Java Virtual Machine itself was the *Princeton Class Loader Attack* [21]. It exploited a combination of flaws—an erroneous implementation of the verifier and the class loading mechanism that implements dynamic loading of classes in the Java virtual machine. The faulty implementation allowed the attacker to overwrite system classes with malicious code. The loaded classes were actually verified as valid while they should have been rejected, and the methods where later on called in place of the overwritten methods. The whole concept of dynamic class loading is one of the fundamental weak spots in the Java security architecture [20].

3.5 Complexity-Based Attacks

The techniques presented in the previous section illustrate some of the documented approaches of attacking a Java Virtual Machine. We consider the complexity-based verifier attack to be the severest of these attacks, because it is enabled by the fundamental design properties of the Java bytecode execution framework, and not by errors in its implementation. The code sequences used in this exploit are *legal* and *correct* mobile code programs and as such not rejectable by the verifier. While Microsoft's .NET platform addresses some of the shortcomings that allow such complexity-based attacks to occur, it still uses fundamentally very similar verification algorithms and is very likely susceptible to similar kind of attacks.

3.6 Complexity of Verification

Regarding the complexity of verification, the analysis of straight-line code is inexpensive, since the abstract interpreter only propagates type information through the instructions and computes the abstract stack state after each instruction.

The runtime of such a data-flow analysis is significantly increased if the code contains jumps, exception handlers, or subroutines, which introduce forks and joins in the control-flow graph. When separate control flows are merged together, an

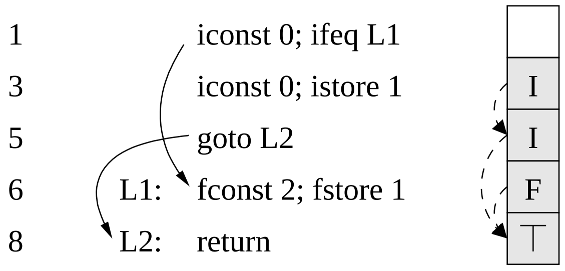


Figure 11: Verification of Java Byte-Code Through Iterative Data-Flow Analysis: The verifier traverses the method from the first instruction to the last. While conditional branch instructions such as *ifeq* are either taken or not-taken by the virtual machine, the abstract interpreter considers both cases at the same time. In this example, the local variable is set to a float in one of the branches, and to an integer in the other. At the merge point (instruction 8), the type of the variable becomes $\top$, because the type of the local variable depends on whether the branch was taken or not. Any attempts by the program to read local variables of type $\top$ would be rejected by the verifier. The example code shown here contains no backward branches, and hence the analysis can be completed in a single iteration. If the taken and not-taken code blocks had been located *before* the *ifeq* instruction (backward branch), the abstract interpreter would have had to iterate over the code a second time to determine the type of the local variable in the merge point.

instruction's predecessors may have different abstract stack or variable types. After merging the state information of the two incoming control flows, the data-flow analysis has to be repeated for all instructions which are reachable from this point in the control flow of the method. For simplicity, the existing Java verifier repeats the entire data-flow analysis for every instruction of a method until there are no more changes.

For average Java programs, the verifier algorithm quickly reaches a fixed point after only a few iterations. For straight-line code or code that contains only forward branches, the verification algorithm terminates already after a single iteration (Figure 11). It is obvious that—*in theory*—the Java verifier could need up to n iterations over the method, with n being the number of instructions in the method. Since for each iteration the verifier might have to visit all instructions, the overall complexity is at least $O(n^2)$.

Such quadratic runtime behavior does not only exist in theory. In fact, simple Java programs can be constructed that expose the worst case scenario in practice. Figure 12 shows a very simple Java program that does nothing but store an integer into a local variable and jump backwards through the code until it finally returns.

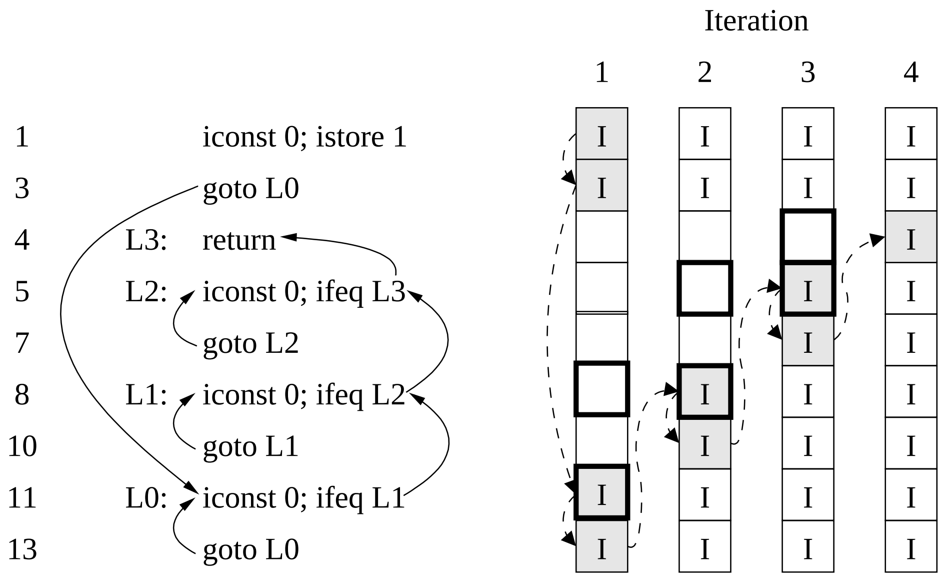


Figure 12: Java Bytecode Program that Takes n iIerations to Verify Using Sun's Standard DFA Verifier Approach: The entry state for each basic block depends on the successor basic block. The type of the first local variable is displayed for each iteration of the DFA. It is initially assumed to be of unknown type and is discovered to be an integer (I) during successive iterations. Shaded boxes indicate a change in the current iteration, framed boxes will be visited in the next iteration.

Studying the verifier algorithm reveals that newly computed type information is forwarded immediately to instructions that come syntactically *after* the current instruction. To instructions that come syntactically *before* the current instruction, the new abstractions will only be forwarded in the next iteration of the DFA. This property is given by the order in which the algorithm iterates over the instructions in each method. Once an instruction has been visited for a particular iteration, it will not be visited again, even if new information about the operand types of that instruction is computed. Thus, if we manage to order N instructions in such a way that each depends on the completion of the verification of the *successor* instruction, we effectively force the verifier to repeat the data-flow analysis N times. For the example in Figure 12, the verifier is forced to perform an iteration for every backwards jump.

The simplistic approach of the traditional Java bytecode algorithm to iterate over the bytecode until a fixed point is reached simplifies the generation of attacks like the one shown in Figure 12, but any other iteration order would also

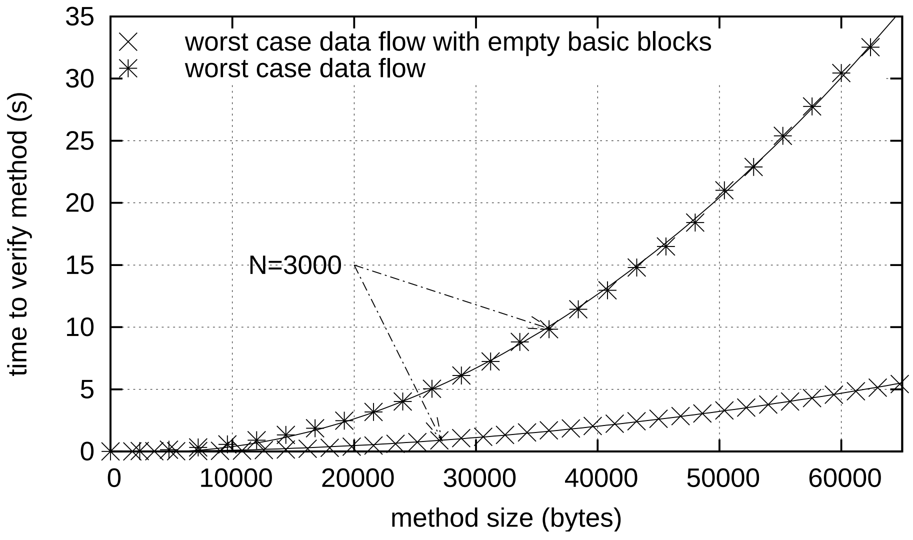


Figure 13: Verification Time for Verifying a Single Method Containing a Worst-Case Data-Flow Scenario: The x-axis indicates the length of the method bytecode in bytes, which is proportional to the number of basic blocks N used to construct the code. The arrows indicate for comparison purposes the code size for path length $N = 3000$.

exhibit a particular (possibly different) worst-case behavior for which a malicious program could be constructed. For any given iteration order for the dataflow analysis, a worst-case ordering problem can be provided that will exhibit quadratic complexity. We intend to work together with makers of existing virtual machines and just-in-time compilers, helping them to identify the worst cases for their specific platform and harden it against this type of attack.

3.7 Exploiting the Worst-Case Behavior

We have measured the verification time for two malicious programs designed to exhibit the worst-case performance of the Java verifier using the Sun Microsystems Java 2 HotSpot Client VM [31].

Figure 13 shows the verification time for a single method containing bytecode with an increasing maximum data-flow path of length N. This time includes only the time it takes the verifier to prove safety. The code is never actually executed or compiled to executable code. The first curve shows the verification time for a worst-case path length problem with empty basic blocks. The second curve in the

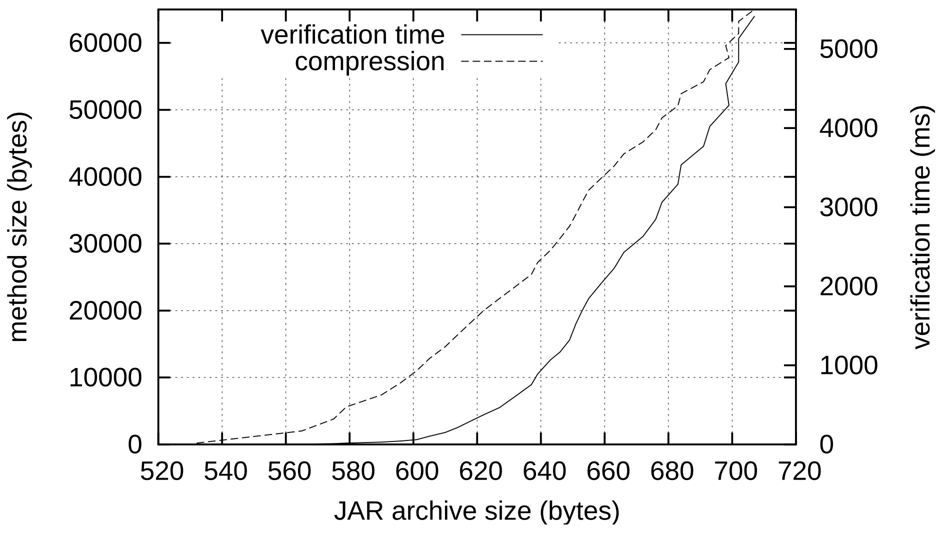


Figure 14: Compression of Constructed Code Examples Using the Standard JAR Archive Format: The code is extremely well compressible as it repeats identical code patterns. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes.

graph shows the maximum flow path problem with some additional code added to each basic block, which further slows down the verifier. Both curves clearly show quadratic growth.

All measurements were taken on a 2.53 GHz Pentium 4 and the Sun HotSpot VM 1.41. The maximum verification time we observed on this machine *for a single method* was approximately 40 seconds. Since the size of method code in Java is limited, this time can not be increased. However, to achieve even longer verification times, an attacker could hide more than just one of these methods in the code. Just including 20 methods instead of one would already increase the verification time to approximately 15 minutes on the test machine we used.

3.8 Weaknesses in the Code Transport Format

The standard JAR archive format used by Java can be used to drastically reduce the apparent size of the malicious code. The code patterns used in the presented scenarios lend themselves for compression due to their very regular structure. Figure 14 indicates the compressed size for different problem lengths N. These code fragments can be compressed very well using the standard JAR algorithm, because each block consists at the bytecode level of exactly the same instructions. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes. The JAR archive format thus represents another example of a well-meant algorithm with appropriate average-case performance, which however exhibits very unexpected worst-case behavior.

We have used the two algorithmic shortcomings described here to construct a malicious applet [30] that disables the Java VM of web browsers for several minutes. The applet is 10kb in size and indistinguishable from regular applet code, because in the end it is a still legal and correct Java program.

Short of disabling Java applets, the user cannot prevent or interrupt the loading of this applet. In fact, existing browsers do not even allow the user to interrupt the verification because the browser implementor never considered the verification time to be costly enough. Some browsers, including some versions of the Microsoft Internet Explorer, allow the verifier to continue the verification silently and continue to hog the CPU in the background even if the user leaves a website containing an applet that takes an excessive amount of time to verify.

3.9 Attacking the Compilation Pipeline

Denial-of-service attacks are not limited to the bytecode verification phase, which is executed early in a bytecode execution framework. Any code transformation algorithm applied to mobile code during its path from a portable bytecode format to natively executable machine code is vulnerable at its point of worst-case complexity. This applies in particular to compiler optimization algorithms, which are traditionally chosen for speed in the average case but not for worst-case performance, and some of which use heuristics to solve problems like graph coloring and instruction scheduling that are known to be known to be NP-complete [14, 38].

An example for such an attackable optimization algorithm is register allocation. Register allocation is an important component of any just-in-time compiler that strives to achieve good code quality. The classic register-allocating algorithm is structured after Chaitin's graph coloring allocator [14, 13]. Many improvements and variants have been proposed [5, 9, 34, 45], but most of this research was focused on improving the average-case performance. Poletto et al. showed that register allocation using graph-coloring has a quadratic worst-case complexity for certain pathological cases [56] and proposed a linear-scan algorithm for register allocation. This algorithm is not guaranteed to find the optimal register allocation for any given problem, but has a linear worst-case performance. To truly harden the virtual machine against complexity-based denial-of-service attacks, however, this principle of trading off some code quality in return for linear time complexity has to be extended to the entire code processing pipeline.

3.10 Countermeasures—And Why They Do Not Work

In contrast to security flaws previously discovered in the Java virtual machine [12], the enabling property for complexity-based attacks on the verifier is an *inherent property* of the algorithm used and not merely some *faulty code* that could be exchanged.

Rewriting the verifier algorithm to iterate over the code in some other order, or the introduction of a work list algorithm, would not significantly improve the situation. Each of these algorithms would still expose quadratic runtime behavior for some worst case scenarios.

However, a number of mitigating factors exist. First, current Java virtual machines limit the code size per method to 65,536 bytes. On high-end desktop systems this limits the maximum verification time we were able to achieve using a single method to approximately 40s. This (probably accidental) ceiling prevents the construction of worst case scenarios with near-infinite verification time.

Further shortening the maximum method length of Java methods is not an option, since long Java methods are not uncommon. Some compilers emitting Java bytecode generate long methods close to the limit defined in the Java specification. This includes some XML transformation tools and parser generators. It would be not surprising if Sun decided to remove the current code size limitation in future versions of the Java Virtual Machine.

It seems unlikely that one could establish a clear set of rules to detect this class of malicious programs. Just rejecting a program because it takes more than a certain number of iterations to verify would be arbitrary. On the other hand, trying to detect patterns such as the ones described in this section would not eliminate the problem—more complex and less obvious examples can be easily constructed. It would also get us back to the *pattern matching* approach used in virus detection tools, something that bytecode verification was supposed to free us from.

The impact of the complexity-based attacked just described can be increased by shipping a large number of malicious methods to the verifier. While this increases verification time by only a linear factor, in conjunction with compressed archives (JAR), verification times in the magnitude of minutes and hours are achievable as shown in Figure 14.

Alternatively, adding resource monitoring to the verification process could be used to counter this attack. However, bytecode verification is deeply embedded into the Java virtual machine. Introducing the possibility to abort a running verification from the outside would require invasive changes to Java virtual machine implementations. Similar to all other previously discussed approaches, resource monitoring introduces arbitrary abort conditions for the verifier and might prevent an important and desirable Java applet or agent to run just because it takes longer than expected to verify the code.

Instead of performing the expensive DFA on the code consumer side, it has been proposed to supply the code consumer with the fixed point of the DFA. The consumer then only has to check whether the supplied fixed point indeed satisfies the data-flow equations, which can be done in linear time. In the case of the K Virtual Machine [72] (KVM) this effect is achieved by annotating the bytecode with stack maps for every point reached by a branch or exception. This annotation can be understood as a very specific case of the more generic proof carrying code approach [54], where a proof generator performs the computationally intensive generation of proofs that are transmitted in form of certificates to a proof checker. The proof checker in turn is able to verify the validity of the code using a certificate in linear time. It is unclear whether Sun will adopt the mechanisms found in the KVM into the general purpose Java virtual machines since adding such annotations would break backward compatibility. Even then, however, the rest of the compilation pipeline would still be vulnerable.

3.11 A New Security Paradigm

As already pointed out, we contend that the only chance to counter attacks that are based on the complexity of certain parts of mobile code execution frameworks is a new security paradigm. Instead of targeting only the safety of certain properties of incoming code, the new paradigm must also take into account the complexity of the whole code path on the client, from verification up to execution.

We are currently investigating possible approaches to harden the Java bytecode verifier against complexity-based attacks. The rationale of embedding a verifier into the Java virtual machine was *to ensure that no unsafe program is ever executed by the virtual machine*. As pointed out above, the worst-case behavior of the data-flow analysis used for verification allows to construct a denial-of-service attack. This vulnerability demonstrates the need for not only *correct* but also *efficient* algorithms when dealing with mobile code. With future software applications moving to Grid- and service-based architectures, in which computations are sent to hosts for execution, these efficient algorithms are going to be essential for system reliability in the near future.

One of our main observation is that with respect to worst-case-based attacks, open source initiatives actually worsen the situation. While in the general case the open-source statement “attack points are easily spotted if thousands of developers around the globe inspect the code” is certainly true, it does not hold with respect to worst-case behavior, which is due to algorithmic properties. Open source reveals the algorithm, making the design of an attack possible. The main insight is that *bugs* can be fixed easily, but *algorithms* are difficult to replace.

As we have shown previously, the code compression format used by Java lends itself to conceal from the user the true size of transported programs. Compression algorithms can also be exploited in many other ways. Clasen used a missing range check in the zlib decompression algorithm to construct PNG images that crash the browser because the decompression algorithm tries to allocate unreasonably large amounts of memory [59]. While we suspect that similar vulnerabilities exist in the JAR format used by Java, this has apparently not yet been studied.

We are currently constructing an ”algorithmic testbed” Java Virtual Machine that can be configured with different variants of critical algorithms. We are also developing a tool to automatically generate Java virtual machine class files that present particular hard to solve algorithmic puzzles. This tool is used in benchmarking existing Java virtual machines, highlighting their potential vulnerabilities, and aiding the removal of such vulnerabilities.

Our aim is to harden the existing Java-based information infrastructure already deployed around the world against such complexity-based attacks. Although no such attacks have yet been reported, they could be very costly in scenarios in which computations are sent to remote servers in the form of ”agents”. We intend to collaborate with the makers of existing virtual machines, making them aware of vulnerabilities and helping them eradicate them.

The scope of this investigation needs to be quite broad: For many code optimizations, well known heuristics exist to speed up their average case performance. However, little to no emphasis has been placed on the worst-case behavior of these algorithms in the context of being a potential security risk. In particular, iterative analyses such as escape analysis, register coalescing, live range splitting, instruction scheduling, and register allocation through graph coloring can have a very poor worst-case performance. Existing just-in-time compiler implementations must be analyzed to identify their weaknesses, and also to provide a framework of code-optimization algorithms with well understood worst-case behaviors.

Our attack exploits the worst-case behavior of one part of the compilation pipeline, the data-flow based verification algorithm. As mentioned above, restructuring this data-flow algorithm will not remove its fundamental property of scaling quadratically for some inputs. In order to inhibit this kind of attacks, one needs to harden *each* step in the pipeline.

For the verifier, we have developed such a hardened algorithm. After performing an initial type check using a superficial type system, it converts the Java bytecode to Static Single Assignment form (SSA) [61, 1], and only then checks the consistency of type flows using the whole Java type system to verify type safety [32]. While this algorithm has a higher *average-case* cost than the standard Java verification algorithm, it has a much better *worst-case* behavior. Namely, all phases beside the SSA construction can be performed in linear time. Many higher-end just-in-time compilers for Java generate SSA anyway at later stages of dynamic code generation. While SSA construction is the main cost in our algorithm, these frameworks can get verification at an *incremental* cost by using our verifier and reusing the constructed SSA. Currently, they perform the standard Java verification *before* starting the actual compilation.

3.12 Section Summary

Future software-application architectures are moving to Grid- and service-based architectures, in which computations are sent to hosts for execution. Soon, these service-based execution frameworks will be omni-present, making the actual network-based execution mechanism invisible to the user. In these architectures, efficient algorithms for each step in the chain from *receiving mobile code* to *compiling it to native code* and *executing it* will be needed to protect against complexity-based attacks. The threat of these subtle denial-of-service attacks has been neglected, apparently because it does not occur in daily average-case use of mobile code. In the case of an unsupervised server at the heart of a service-based framework, however, having the framework verifying, analyzing, compiling, and executing many mobile-code programs in parallel will make each and every phase in the framework vulnerable to complexity-based attacks.

We therefore advocate a new security paradigm based on a complexity-hardened infrastructure. With the currently widespread mobile code execution frameworks in place, there is no quick fix to this problem. Instead, we will need to rethink the architecture of those systems—while current systems place verification as a hurdle for incoming code and after that use fine tuned algorithms that have been selected for their average case behavior, we will need to construct systems where each step has a provable *worst-case* behavior. Until these systems are in place, open source code development actually worsens the situation. Instead, security by obscurity actually works.

4 SSA-Based Java Bytecode Verification

Mobile programs can be malicious. A host that receives such mobile programs from an untrusted party or via an untrusted network connection will want a guarantee that the mobile code is not about to cause any damage. To this end, the Java Virtual Machine (JVM) pioneered the concept of *code verification*, by which a receiving host examines each arriving mobile program to rule out potentially malicious behavior even before starting execution. This analysis is necessary since the locations of temporary variables in the Java virtual machine are not statically typed. If verification is successful, then the *original* bytecode is forwarded to the JVM's execution component, which may be an interpreter or a just-in-time compiler. Specifically, beyond the result denoting whether or not verification was successful, all other information computed by the verifier is discarded and is not passed onwards. In many cases, this results in a duplication of work when a just-in-time compiler subsequently performs a very similar data-flow analysis all over again.

In this section, we give a brief overview of an alternative verification mechanism that avoids such duplication of work. Instead of verifying Java Virtual Machine Language (JVML) bytecode directly, we annotate it in such a way that the flow of values between instructions becomes explicit rather than going through the operand stack and then transform the annotated bytecode into Static Single Assignment (SSA) form [18].

Verifying programs in SSA significantly reduces the number of points in the program that have to be type-checked, because only producers and consumers of values are verified. Inter-adjacent transitions of a value through stack and registers are no longer verified explicitly. This integrated approach is more efficient than traditional bytecode verification but still as safe as strict verification, as overall program correctness can be induced once the data flow from each definition to all associated uses is known to be type-safe.

Our benchmarks indicate that the aggregate time required for transforming JVML into SSA and verifying the program in this representation is still less than the time needed for performing the standard verification algorithm directly on JVML. Our approach imposes no overhead for methods that will be interpreted without just-in-time compilation, because SSA-based verification is still overall faster than the traditional verifier.

The remainder of this part of our report is organized as follows: Section 4.1 gives a brief overview of the traditional Java bytecode verifier and introduces SSA-based verification. Section 4.2 compares the performance of our method to that of Sun's standard verifier. Section 4.3 discusses related work and Section 4.4 concludes and points to future work.

4.1 Verification in Static Single Assignment Form

This section introduces a subset of JVML, briefly describes traditional Java bytecode verification, and discusses the abstraction used in our approach as well as our novel verification method.

$$\begin{array}{rcl} \textit{instruction} & ::= & \textit{core} \mid \textit{dataflow} \\ \textit{core} & ::= & \texttt{iconst\_}n \mid \texttt{lconst\_}l \mid \texttt{iadd} \mid \texttt{ladd} \mid \texttt{ifeq}\ L \mid \texttt{return} \\ \textit{dataflow} & ::= & \texttt{pop} \mid \texttt{dup} \mid \texttt{dup\_2} \mid \texttt{istore\_}x \mid \texttt{iload\_}x \mid \texttt{lstore\_}x \mid \\ & & \texttt{lload\_}x \end{array}$$

Figure 15: Instructions in JVML$_{\mathcal{S}}$. The Arguments n, l, x, and L Must Fulfill the Conditions $-1 \leq n \leq 5, l \in \{0, 1\}, x, L \in \mathbb{N}$.

4.1.1 JVML$_{\mathcal{S}}$

Figure 15 shows the grammar for JVML$_{\mathcal{S}}$, a subset of JVML which we use here for illustration purposes. We split the instruction set in *core* instructions and *data-flow* instructions. Core instructions operate on values stored on the operand stack, while data-flow instructions only facilitate the flow of values between core instructions by manipulating the state of the operand stack and exchanging values between operand stack and variables.

Values are produced by core instructions and can be consumed by other core instructions. During the lifetime of a value it can reside on the operand stack or in variables and in multiple locations at the same time. Data-flow instructions neither produce nor consume values, they merely transport values between stack locations and variables. [2]

4.1.2 Java Bytecode Verification

JVML instructions can read and store intermediate values in two locations: the operand stack and local variables. These locations are ad-hoc polymorphic in that the same stack location or local variable can hold values of different types during program execution. Verification ensures that these locations are used consistently and intermediate values are always read back with the same types that they were originally written as.

Verification also ensures control-flow safety, but this is a comparatively trivial task. Conversely, verifying that the data flow is *well-typed* is rather complex. The Java virtual machine bytecode verifier [48, 78] uses iterative data-flow analysis and an abstract interpreter for JVML instructions. Unlike in the Java virtual machine, the stack cells and local variables of the abstract interpreter store *types*, rather than

[2]Even though it consumes a value, the *pop* instruction is a data-flow instruction, since it merely manipulates the stack such that the topmost value can no longer be used.

values. From the perspective of the verifier, Java virtual machine instructions are operations that execute on types.

JVML verification works at the method level. With a co-inductive argument it follows that if every method is verifiable, the whole program is verifiable, too. In the rest of this report, we use program and method interchangeably.

The central responsibility of the Java bytecode verifier is to check that stack locations and local variables are used in a type-safe manner. This is the case if the definitions and uses of values have compatible types. To ensure this, the verifier algorithm has to determine the types of all stack locations and variables for each instruction.

4.1.3 Abstractions

In JVML, there is no obvious link between the definition of a value and its uses. However, even if definition-use chains were available for each value in a JVML program, it would still be impossible to verify a Java program in a single pass by comparing the type of each definition with its uses. The reason for this becomes more obvious if we consider how we categorized the instructions of JVML$_{\mathcal{S}}$. Only *core* instructions define and use values. *Data-flow* instructions merely facilitate the flow of values between core instructions. For Core instructions the expected types of any consumed operands and the types of any produced values are always known statically. In contrast, data-flow instructions are polymorphic. In general, it is not possible to determine the type of the value produced by a data-flow instruction without knowing the type of its operands. The result type of a `dup` instruction, for example, depends on the type of the value on top of the stack.

While local variable access instructions such as `iload`$\_x$ suggest stronger static typing, this works for scalar types only. In the Java virtual machine, object references are written and read from local variables using `astore`$\_x$ and `aload`$\_x$, and data-flow analysis is still necessary to determine the precise type of the variables accessed.

The rationale of our approach is to replace the stack and local variables by a register file, and to redefine the dynamic semantics of instructions to actually work on these registers. This replacement allows us to transform the stack based code into SSA and to perform type checking only between the definitions of values and their actual uses. We abstract each instruction in a program to a tuple consisting of the depth of the stack before that instruction is executed, a mapping from stack cells and local variables to the instructions that define them, the set of stack cells and local variables the instruction reads and writes, as well as a map from stack cells to the values that reside in them. The main contribution of these components

PC	Instruction	StackDepth	Stack						Vars	
			0	1	2	3	4	5	0	1
1	`lconst_0`	0	`L`	`L`’						
2	`lconst_1`	2	`L`	`L`’	`L`	`L`’				
3	`iconst_1`	4	`L`	`L`’	`L`	`L`’	`I`			
4	`ifeq L`	5	`L`	`L`’	`L`	`L`’				
5	`dup_2`	4	`L`	`L`’	`L`	`L`’	`L`	`L`’		
6	`ladd`	6	`L`	`L`’	`L`	`L`’				
7	`L:` `ladd`	4	`L`	`L`’						
8	`lstore_0`	2							`L`	`L`’

Figure 16: An Example Program and the Abstraction for Stack and Variable States. Each Instruction is Labeled with the Stack Depth Prior to the Execution of that Particular Instruction. `L` stands for `LONG`, `L`’ for `LONG`’, and `I` for `INT`.

is to allow the dynamic semantics to work on a register file and to enable the transformation of the code into SSA *before* verification.

4.1.4 Algorithm

The goal of our approach is to avoid an up-front iterative data-flow analysis to verify JVML. Instead, the JVML code is annotated so that the flow of values between core instructions becomes explicit instead of relying on an operand stack. This enables us to eliminate all data-flow instructions from the code after SSA construction. These instructions are no longer needed because they only facilitate data flow, but do not actually compute anything. Once the code consists of core instructions only and is in SSA form, it is possible to perform type-safety checks by directly relating the type of each definition with the corresponding uses (*definition-use verification*).

For a small example program, the result of the annotation step is shown in Figure 16. Each instruction is annotated with the current stack depth before the instruction is executed. Using these annotations and the dynamic semantics of $\text{JVML}_{\mathcal{S}}$, instructions no longer depend on the stack to connect operands to their definitions. Values on the stack are labeled relative to their distance to the bottom of the stack. The value produced by an `iconst` instruction executed on a previously empty stack, for example, would be labeled with 0, because it is currently at the bottom of the stack. This labeling permits to resolve stack references without actually maintaining a stack data structure. An `iconst` instruction annotated with *sd* $= 0$, for example, always writes its result to stack cell 0. In unannotated JVML the stack cells receiving the produced value would depend on the state of the dynamic stack at that point in the program.

Following the JVML machine model we split long integers into two halves. Thus, instructions operating on long integers push and pop pairs of values onto and from the operand stack. Correspondingly, for each definition of a long integer two values are defined, one for the bottom half (type `LONG`), and one for the top half (type `LONG`').

After the annotation phase, our verification algorithm first computes the Iterative Dominance Frontier (IDF) [66] for all definitions of values, that is values written into stack cells or local variables. Each reachable instruction in the program is visited in dominator-tree order and all references of core instructions to stack cells and local variables are resolved to SSA-names. Data-flow instructions do neither produce nor consume any values and are eliminated through copy propagation.

After transformation into SSA and copy-propagation, we can perform the actual type-checking. Similar to type inference performed by the traditional verifier, the type of ϕ-nodes is the common supertype of each definition the ϕ-node refers to (ϕ operands), while regular core instructions always define a value with a distinct type. These can be matched to their respective uses in a single sweep over the program in linear time.

Type-checking is performed lazily in the sense that only the minimal number of instructions is checked to ensure overall type-safety while for dataflow instructions only the proper data flow is guaranteed. Considering only the dynamic semantics, the data flow verified is obviously equivalent to the data flow that would have resulted by interpreting the original JVML program. However, since data-flow instructions have been eliminated, some of the restrictions enforced by their static semantics do no longer apply. The following JVML program, for example, will be rejected by the Java verifier, but is valid in our SSA-based dialect:

```
1: lconst_0
2: istore_1
3: iload_1
4: lstore_2
```

In this example, in Line 1 a long integer is pushed onto the stack as a pair of halves $(\texttt{LONG}, \texttt{LONG}')$. Partially storing the long integer in an integer register (Line 2) is rejected by the traditional verifier. In contrast, since our verifier does not consider the typing rules of data-flow instructions, it accepts this code fragment, because the $(\texttt{LONG}, \texttt{LONG}')$ pair pushed in Line 1 is restored on the stack before it is used in

Line 4. It is important to note that this program, while rejected by the Java virtual machine, is perfectly safe when executed.

Due to space limitations, we are unable to elaborate on how to verify exceptions, arrays, and object initialization and refer to our technical report [33] instead.

4.2 Benchmarks

To evaluate the performance of our SSA-based verifier, we have implemented a prototype verifier based on the algorithm presented here. Our prototype inlines subroutines before verification. In order to arrive at a fair comparison with Java's standard verifier, we use the same modified Java code with inlined subroutines also for the JVML verification benchmarks. Our rationale behind this is that the subroutine construct in Java is obsolete and will probably be removed in future versions of the Java virtual machine. Furthermore, our current algorithm depends on the fact that the control-flow graph can be recovered quickly from JVML code. In the presence of subroutines, this is not always the case as returning edges from subroutines are not explicit.

As a comparative benchmark, we compare the total runtime of our SSA-based verifier to the runtime of Sun's DFA-based verifier. In both cases, we use the preverify tool shipped as part of Sun's KVM [52] to inline all subroutine calls before measuring the actual verification times. Both verifiers are implemented in C and use the same underlying framework to read and represent Java class files.

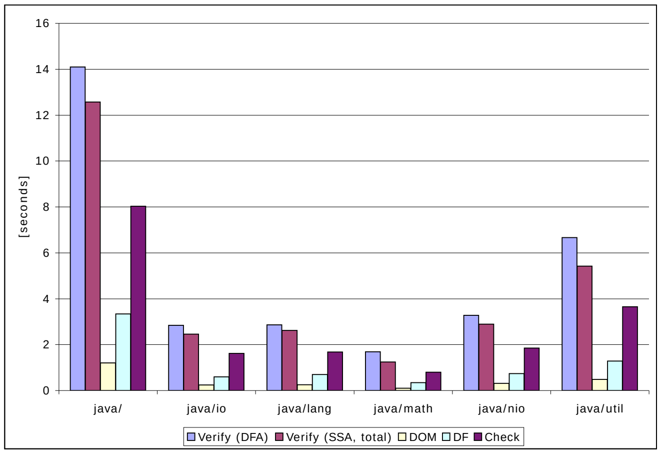
To eliminate any cache effects and to compensate for timing errors, both verifiers are run one hundred times on each method from the test set. There currently is no established set of benchmarks to test the performance of verifiers. Benchmark suites such as SPECjvm [65] are designed to evaluate the performance

	# of	**method size**		**stack depth**		**local variables**	
	methods	**ø**	**max**	**ø**	**max**	**ø**	**max**
java/*	6490	41.36	4065	2.74	14	2.47	37
java/io	1213	38.12	1295	2.39	8	2.35	15
java/lang	1336	38.41	4065	2.32	10	2.17	37
java/math	405	72.67	3041	3.16	8	3.73	29
java/nio	2096	26.80	417	3.05	11	2.31	15
java/util	2359	49.21	2916	2.64	14	2.62	25

Figure 17: Characteristics of the Test Set Used to Compare the Runtime of the SSA-Based Verifier With the Runtime of the Traditional Verifier.

of code execution, *not code verification*. Thus, we have decided to use various parts of the Java Runtime Libraries (JDK 1.4.2) as a test set. Figure 17 list some characteristics of the used classes. All measurements were conducted on a Pentium4 2.53GHz CPU with 512MB of RAM, running under RedHat Linux 9.

Figure 18 compares the total runtime of the traditional DFA-based verifier with our SSA-based verifier. Verification in SSA-form is approximately 15% faster than the traditional algorithm when comparing the total runtime. Not considering the time spent to calculate the dominator relation and the dominance frontier, SSA-



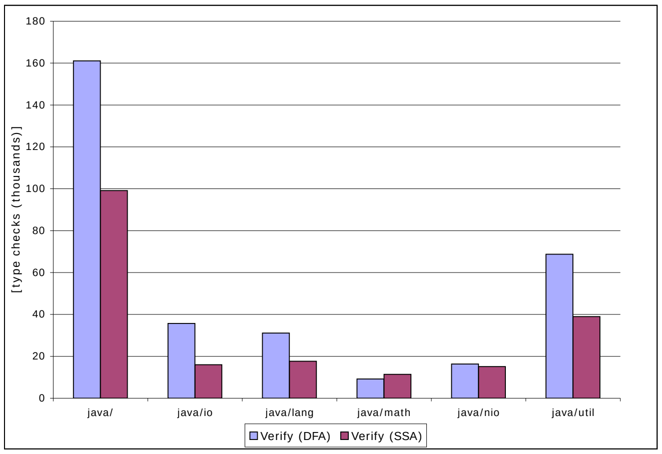


Figure 18: Comparison of the Total Runtime and the Number of Instructions that have to be Type-Checked for the Traditional DFA-Based Verifier vs. the SSA-Based Verifier.

based verification is approximately 45% faster. The total number of instructions that has to be type-checked in the case of SSA-based verification is roughly 38% less than for the traditional verifier. The only noteworthy exception is *java/math*, which actually requires slightly more instructions to be type-checked in SSA form. This is caused by our treatment of the `LONG` and `DOUBLE` types, which we split in two halves while the traditional verifier can treat them in a single step.

4.3 Related Work

In addition to the informal description of the Java virtual machine [48], a number of formal specifications of the JVML and its verifier have been proposed [27, 47, 67]. In this context, subroutines are of particular interest and several type systems have been proposed for them [55, 57, 68]. All these approaches have in common that they rely on some form of iterative data-flow analysis [47, 58] to decide type-safety.

Proof-carrying code (PCC) [54] addresses this problem by relieving the code consumer of the burden to verify the code. Instead, the code producer computes and proves a verification condition. The code consumer recomputes the verification condition and checks whether the attached proof is valid. PCC can even be used to prove safety properties of machine code. SSA-based verification, in contrast, is limited to mobile code formats such as Java, but has the advantage that it only requires the actual code as input, and no additional information such as proofs.

The split verifier approach [70], based on the idea of Lightweight Bytecode Verification [60], applies the PCC idea to Java bytecode. A *preverifier* annotates the JVML with the fixed-point of the data-flow analysis otherwise performed by the JVM during class loading. For annotated class files the verification is reduced to confirming that the annotation is indeed a valid fixed-point. Just as in the case of Necula's PCC, the annotations enlarge the overall size of class files, while our approach does not rely on any additional annotation.

Similar to the split verifier, the verifier for Java smart cards [46] reduces the burden on the verifier through offline bytecode transformation. A preprocessor tool ensures that the Java stack is empty after every branch instruction and that all registers are mono-typed. In contrast to our approach, the Java smart card verifier fails for Java class files which have not been processed this way.

Inherently safe mobile code representation formats such as SafeTSA [2] eliminate the need for verification as mobile code is stored in a self-consistent format that cannot represent anything but well-formed and well-typed programs. Just like PCC, such formats have a systematic advantage over SSA-based verification, but require abandoning the existing Java class file format, which is not always acceptable. Our approach and SafeTSA have in common that they both make the code available to the just-in-time compiler in SSA-form, which can be used to speed up code generation.

SSA-based representations have been used in several approaches to compilation of bytecode. Marmot [24] is a research platform for studying the implementation of high-level programming languages. The main difference to our work is that Marmot only accepts verifiable programs. This property of the input program allows to make certain assumptions on properties of the code, e.g. about the types of local variables and stack entries. Similar to our work, Marmot inlines subroutines to avoid complex encoding as normal control flow similar to Freund [28].

As Kelsey and Appel have observed [4, 43], there is a close relation between SSA form and functional programming. Therefore, the work of League et al. [44] is directly related to our work. λJVM, a functional representation of Java bytecode, makes data flow explicit, just like our work. They also split verification up in two phases, one during the construction of λJVM code, and a simple type checking later. However, they initially perform a regular data-flow analysis to infer types for the stack and local variables at each program point. This is in contrast to our approach, were the reason for splitting the verification in two phases is exactly to avoid the initial data-flow analysis.

4.4 Section Summary

Existing JVML verifiers perform substantial data-flow analysis but do not preserve the results of this analysis for subsequent code generation and optimization phases. We have presented an alternative verifier that not only is faster than the standard Java verifier, but that additionally computes the Dominator Tree and brings the program into Static Single Assignment form. As a result, the respective computations need not be repeated in subsequent stages of the dynamic compilation pipeline. Since our algorithm has an overall lower cost than traditional Java bytecode verification, this essentially makes an SSA representation available “for free” to the virtual machine, reducing the cost for just-in-time compilation.

In the larger context of verifiable mobile code, our results indicate that verification should not be practiced in isolation “up front”, but integrated with the rest of the client-side mobile code pipeline. Hence, we expect our approach to be applicable to other mobile-code systems besides the Java virtual machine, such as Microsoft’s .NET platform [17].

Our work is also relevant for all existing Java virtual machine implementations which already use SSA internally for code optimization. If a virtual machine already has means to translate code into SSA, having an “up front” data flow based verifier is simply redundant. We have shown that it is possible to delay type checking and to first transform the program into SSA. In fact, our algorithm is the first documented approach to safely translate Java code into SSA without any prior data-flow analysis and verification.

In the future, we plan to examine how subroutines could be supported in our framework. While subroutines are rapidly disappearing from JVML, they are still interesting from an academic perspective. They reinforce the question whether and how an SSA-based representation can be obtained for polymorphic code in which not all control-flow edges are explicit.

We are also interested in exploring *structural* SSA-annotation of JVML code. For this, JVML code is rearranged in such a way that a specific structure-aware SSA-based verifier can infer the final SSA-form of the code without actually calculating the Dominator Tree and Iterative Dominance Frontiers. As the code is still expressed in pure JVML, it is fully backward compatible with existing virtual machines and does not require any additional annotations. While the rearranged code is likely to be less compact than its original form, this scheme will further reduce the required verification effort.

5 Conclusion

This research project has made three important contributions that are likely to have a lasting impact on the design of mobile-code systems. By demonstrating how code can be generated *inside the network*, we believe to have created a blueprint for future “soldier radio” and similar data networks with resource-limited handheld terminal nodes.

By exposing the vulnerability of verification-based mobile code formats such as Java to *attacks based on the complexity of the verification mechanism*, our work puts into perspective the claims of the open-source community. We hope that this will effect a rethink of current strategies that optimize for the average case, rather than for the worst case.

Lastly, we have invented a new *mobile-code verification algorithm that benefits subsequent code optimization*. We expect others to adopt our approach, which has no apparent disadvantage but many obvious advantages.

ONR Final Report

Co-PI: Brett D. Fleisch, Professor

Department of Computer Science and Engineering
University of California, Riverside

Grant Period: 12/1/2002~4/30/2004
Grant No.: 00767-004

I. List of Written Publications

1. Yougang Song, Ying Xu and Brett D. Fleisch, BRSS: A Binary Rewriting Security System for Mobile Code, *submitted to* 25th IEEE International Conference on Distributed Computing Systems (ICDCS2005), Columbus, Ohio, June 6-9, 2005.

2. Ying Xu and Brett D. Fleisch, NFS-cc: Tuning NFS for Concurrent Read Sharing, the International Journal on High Performance Computing and Networking (IJHPCN), Inderscience Publishing, issue 3, 2004

3. Yougang Song and Brett D. Fleisch, Sandboxing Mobile Code from Outside the OS, *in the 19th ACM Symposium on Operating Systems Principles,* Work in Progress Session, The Sagmore, Bolton Landing (Lake George), New York, 2003.

4. Yougang Song and Brett D. Fleisch, Rico: A Security Proxy for Mobile Code, Journal of Computers and Security Elsevier Advanced Technology, Elsevier Press, Volume 23, Issue 4, 2004, pp. 338-351.

5. Ying Xu and Brett D. Fleisch, Cooperative Caching in Linux Clusters(pdf), Proceedings of the ClusterWorldConference and Expo 2003, San Jose, CA, Jun 23-25, 2003, San Jose, CA.

II. Personnel

Co-PI: Brett D. Fleisch (Associate Professor)

Dr. Fleisch currently has assumed a position with the National Science Foundation (NSF) in Arlington, VA as Program Director for Distributed Systems and Operating Systems. He remains an associate professor at the University of California faculty while on assignment to NSF. Dr. Fleisch received his B.A. degree in Computer Science at the University of Rochester, his M.S. degree in Computer Science at Columbia University, and his Ph. D. degree from the University of California, Los Angeles. He is a member of the ACM, IEEE Computer Society, and USENIX. Dr. Fleisch's research areas include: distributed computing, operating systems, workstation environments, large scale computing systems, operating systems software engineering metrics, security, reliability, heterogeneity, and software quality assessment and power management for computing clusters.

Employees:

Peter H. Froelich is currently a Postdoctoral Scholar and lecturer in the Department of Computer Science and Engineering at the University of California, Riverside. He received his Diplom-Informatiker degree from the Munich University of Applied Sciences, Germany, his master's degree in Information and Computer Science from the University of California, Irvine, and his doctorate in Information and Computer Science from the University of California, Irvine. His research focuses on the fuzzy intersection of software engineering, programming languages, and computer systems.

Ying Xu is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the University of California, Riverside. He received his B.E. degree at Tianjin

University, China. His research area focuses on operating systems, cluster systems, distributed file systems and mobile code systems.

Yougang Song is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the University of California, Riverside. He received his B.E. degree in Electrical Engineering from Shandong University of Technology, China, his M.E degree in Computer Engineering from Chinese Academy of Sciences, China, and his M.S. degree in Computer Science from University of Texas. His research area focuses on operating systems, mobile code security, and distributed file systems.

David Watson is an undergraduate employee at the University of California, Riverside.

III. List of Papers Presented

1. Yougang Song and Brett D. Fleisch, Sandboxing Mobile Code from Outside the OS, *in the 19th ACM Symposium on Operating Systems Principles*, Work in Progress Session, The Sagmore, Bolton Landing (Lake George), New York, 2003.

2. B. D. Fleisch, Grand Challenges in IT Security and Assurance, *First International Workshop on Frontiers of Information Technolgy*, Islamabad, Pakistan, December 23-34, 2003.

3. Ying Xu and Brett D. Fleisch, Cooperative Caching in Linux Clusters, Proceedings of the ClusterWorldConference and Expo 2003, San Jose, CA, Jun 23-25, 2003, San Jose, CA.

Sandboxing Mobile Code from Outside the OS

Yougang Song Brett D. Fleisch
{ysong, brett}@cs.ucr.edu
University of California, Riverside (UCR)

(Extended Abstract)

INTRODUCTION

Today's highly connected computer systems are more frequently exposed to code originating from various sources. Malicious code, or even trusted code, may compromise security. Even the best-intentioned security patch downloaded from the Internet may violate security policies an administrator has established. Execution-time methods to mediate access to mobile code downloaded from the Internet, such as a kernel reference monitor or Java sandboxing can be costly, inefficient and error prone [1]. A global security mechanism requires code originating from one system execute safely on another system when there is no established trust relationship between the systems.

At UCR we designed a novel security infrastructure called Rico [1]. Rico interposes itself between a client system that requests mobile code and the remote system providing the code, monitoring and securing untrusted code by rewriting it (as shown in Fig. 1). Rico leverages the Inline Reference Monitor (IRM) binary rewriting technique. A load-time rewriter (Rico supports PoET, developed at Cornel University) merges the security policy into the mobile code binary. In addition, Rico supports features like mobile code management, policy management, policy acquisition and reuse [1].

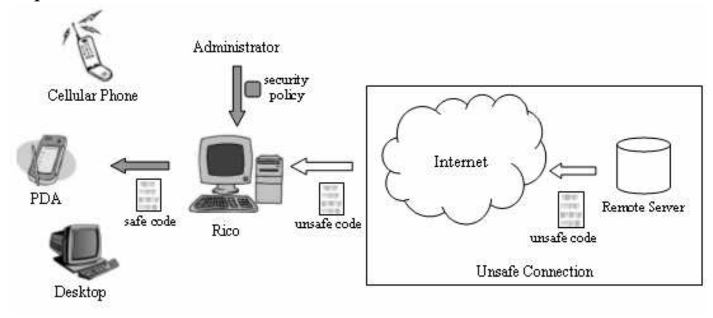


Figure 1. Sandboxing mobile code from outside the OS

Rico functionality can be placed at Internet access points such as proxy servers to isolate end user from security problems. Different from many content security systems, which merely focus on detecting viruses or malicious code by comparing the received binary for virus signatures or a list of dangerous activity patterns, Rico is based on the idea of *sandboxing* the code. Administrators can confine a mobile code's activity by defining specific security policies suitable for their particular domain. These policies are merged into the mobile code so that the code will strictly adhere to these security constraints.

In this work, we integrate Rico into Apache, which is currently the most popular proxy server deployed. The resulting system provides security for mobile code downloaded to end users. Java applets and (in the future) OS patches can both be rewritten to guarantee better security. Our preliminary experiment results show that such a system adds little performance overhead.

INTEGRATING RICO WITH APACHE

The Apache filters mechanism processes a Web document for several pieces of information. Each piece of information will be processed at one filter before it is passed to the next. We thus designed a security filter before all the other output filters; the security filter observes the binary code in the data flow and sends a notification to Rico. Rico runs as a service daemon and rewrites the binary code with the preloaded security policies that pertain to Applets or patches of this kind. The rewritten binary code is then passed to the next filter, which caches the code in the proxy cache (if cacheable) and sends it to the client. Subsequent requests for the same binary code will be satisfied from the proxy cache without the intervention of the security filter. Apache handles cache coherence so cached data remains current.

PRELIMINARY RESULT AND FUTURE WORK

We evaluated the performance overhead of Rico combined with Apache. The experiment was performed on a Dell Dimension 2450 with 2GHz Pentium4 CPU, 640MB RAM and Mandrake Linux 9.0 installed. We rewrote the codes with the policy *limitMem* and *readDir* respectively. limitMem limits the amount of the memory that the application is allowed to use and has the worst performance overhead in our measurements [1]. readDir prevents the codes from accessing a specific directory. Fig. 2 shows the performance as compared with each binary file's size. The data label above each bar refers to hit times, which is the number of events that PoET need to add monitoring code to. For readDir, there is no hit and the time merely reflects the time that PoET takes to scan through the binary file. For limitMem, there are different amount of hit times for each code and the time reflects the time that PoET takes to scan and rewrite the code.

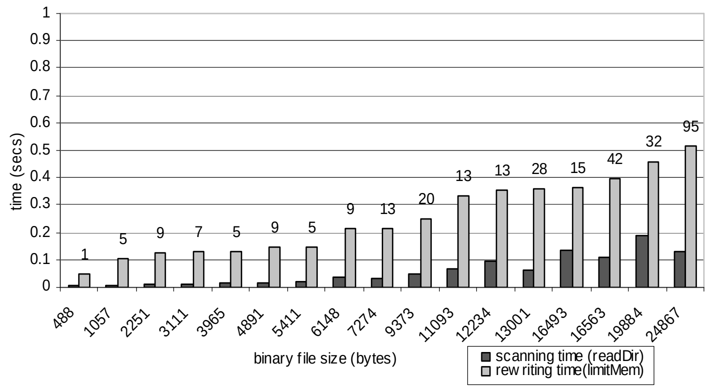


Figure 2. Rico performance evaluation

In [1], we have shown that both the scanning and rewriting time are generally a linear function of file size; the rewriting time is also linearly proportional to hit number. Fig. 2 shows that the Java initialization time from PoET startup [1] is minimized and the performance overhead increases gradually as the file size and hit times increase. However, selecting a default set of policies that can generally satisfy an end system's security requirements can be challenging. Future work will evaluate this more extensively.

[REFERENCE]: Yougang Song, Brett D. Fleisch, Rico: A Security Proxy for Mobile Code, *to appear in the Journal of Computers & Security*, ELSEVIER Advanced Technology, 2003.

BRSS: A Binary Rewriting Security System for Mobile Code

Yougang Song Ying Xu Brett D. Fleisch
Distributed Systems Laboratory
Department of Computer Science & Engineering
University of California, Riverside
{ysong, yxu, brett}@cs.ucr.edu

Abstract

In recent years, binary rewriting techniques have been advanced to allow users to specify a richer set of security policies and to enforce them directly on mobile code. Such techniques have their advantages over traditional mobile code security solutions. However, problems arise concerning how we can use binary rewriting to achieve better security and how code rewriting overhead affects system performance. In this paper, we address these questions by presenting our Binary Rewriting Security System (BRSS), which is a prototype that integrates binary rewriting with proxy caching techniques.

The contributions of this paper are: (1) we designed a general framework that supports a variety of binary rewriters, while at the same time minimizes the rewriting overhead and provides efficient management for mobile code and associated security policies and (2) we built a prototype of a binary rewriting proxy by integrating the binary rewriter with caching proxy; (3) we conducted a study on Internet traffic focusing on the traffic characteristic of JAVA applets; (4) we did a comprehensive performance evaluation of our prototype system. The results show that adding such a system to the proxy server doesn't significantly affect overall performance and the overhead added by binary rewriting will easily be amortized during caching.

1. Introduction

Mobile code is the code that can be transmitted across the network and executed on the recipient. JAVA applets are the most common form of mobile code. These applets are usually embedded in web pages and automatically downloaded and executed in the users' web browsers. Failure to properly secure mobile code may cause security threats to the host system, such as damage to the file system, excessive use of resources and leakage of private information [1]. Not only could untrusted code be created with malicious intention, but also it could result unintentionally. Specifically, programming errors can be exploited by attackers in mobile code [2]. The buffer overflow problem, in which the overflowed content can overwrite existing data, instructions, returned address in the memory, is such an example.

Traditional solutions to mobile code security have been examined from the operating system perspective. They employ a host security system that monitors a mobile program's behavior and determines whether resource requests are granted or rejected access according to some predefined security policies or administrative policies. For example, using a kernel reference monitor [7], the kernel is the only software trusted to access the critical system data structures. Consequently all access requests to critical data is mediated by the kernel through system calls. In JAVA [8], however, security is enforced by three critical components. The *byte-code verifier* checks the untrusted code to make sure that it does not violate memory safety properties. The *applet class loader* ensures that the JAVA classes are separated in different name spaces and tagged properly with security information such as the code's origin and digital signature. The *security manager* acts as the reference monitor to enforce run time checks.

Binary rewriting techniques have two major advantages over the traditional solutions. First, it supports a richer set of security policies. Traditional host security systems provide general security rules that enforce access control, but it is hard for them to enforce a security policy such as "no sending messages after reading specific files". However, binary rewriters have the flexibility to insert arbitrary security checks into the mobile code. It is possible for them to add an Inline Reference Monitor [5] to monitor the execution status of the mobile code. Consequently binary rewriters can be easily customized to support more advanced security policies. Second, a binary rewriting system doesn't rely on the host security system to enforce security. The inline reference monitor inserted into the mobile code is self-contained. It observes the execution of the mobile code and terminates it when it is about to violate a security policy. Because running a host security system requires much more resources, this characteristic of binary rewriting is particular useful to the systems that have tight constraints on power consumption and memory capacity such as Personal Digital Assistants (PDAs) and mobile sensors.

However, binary rewriting systems [22, 23, 6] were designed to work with standalone computers. The administrative overhead of configuring security policies and managing the rewritten code prevents them from being used in large scale. The performance overhead in this case not only includes the known rewriting overhead but also the overhead caused by system initialization. For example, in order to run PoET, the JAVA virtual machine should be initialized first and then the security policies should be parsed and loaded into memory. Also, such system usually lacks a convenient application and management interface for security administrators. Tools such as PoET were merely command line oriented.

We solved these problems by integrating binary rewriting into web proxy servers and designed BRSS (Binary Rewriting Security System). BRSS gives system administrators centralized control of security polices and a uniform way to enforce security policies at the level of administrative domains through proxy servers. All applets passing through the proxy server are automatically rewritten and guaranteed to be secure before they reach the recipients. There is no specific security configuration required at the recipients' side. BRSS eliminates the binary rewriter's initialization overhead by running them in the background as a multithreaded process. To reduce the overhead, we designed BRSS to cache the rewritten applets. All subsequent requests for the same applets are served directly from the cache without rewriting overhead. According to our comprehensive performance evaluation, under the realistic JAVA applet traffic model, the caching of rewritten applets greatly amortizes the rewriting overhead and thus BRSS doesn't add significant performance overhead to the proxy server.

The rest of paper is organized as follows: Section 2 gives a short survey of binary rewriting techniques to security. Section 3 describes BRSS, its architecture and implementation. Section 4 gives the performance evaluation. Section 5 discusses related work, followed by conclusions in Section 6.

2. Binary Rewriting Techniques to Security

A binary rewriting system transforms a binary program into a different but functionally equivalent program [3]. Because it requires no knowledge of the source code, binary rewriting has been widely used in the area of code migrations across different processor architectures, performance instrumentation and program optimization, such as optimizations on code speed, size and power consumption [4,9,10,11,12,13,16]. Application of binary rewriting techniques in computer security area mainly focused on dealing with commonly seen and dangerous attacks, such as the buffer overflow problem [18]. Before real binary rewriting security techniques appears, many works are proposed based on compile time analysis and transformation such as RAD [14], which protects buffer overflow attacks by adding protection code into the prologue and epilogue of the program at the compile time; Or based on run-time interception and checking such as Libsafe [21], which is based on a dynamically loadable library that intercepts all function calls made to library functions that are known to be vulnerable; Or the sandboxing techniques such as Software Fault Isolation (SFI) [24], which uses the idea of Address Sandboxing to enforce system security.

The Binary-Rewriting RAD [15] extends the work of RAD and first applies the security directly on binary code without requiring access to program source code, symbol tables or relocation information. It uses a combined disassembly techniques to identify the boundary of every procedure in the input program. The protection code is appended to the end of the original binary. It inserts the code at the function prolog to save a copy of the return address and the code at the function epilog to check the return address on the stack with the saved copy. Some instructions at the function prolog and epilog are replaced by a JMP instruction to redirect the control to the inserted code at a function's prolog and epilog. Purify [25], detects run-time memory leaks and access errors by inserting checking instructions directly into the object code and before every load or store. To detect memory access errors, Purify maintains a state code for each byte of memory and a run-time check is enforced by the checking instructions whenever time the program makes memory access. To make binary analysis and rewriting efficient, SELF [21] proposed a transparent security enhancement to ELF binaries by adding an extra section. The extra section contains information specifically needed for binary analysis and hence it is convenient to perform many security-related operations on the binary code.

Comprehensive binary rewriting security systems appeared when binary rewriting techniques were used to enforce security policies specified by host system requirements. The benefit provided by such security systems is the flexibility to enforce security policies that are not supported by the standard security mechanisms [26]. Notable such systems are Naccio [22], SASI [23] and PoET [6]. Naccio allows the expression of safety policies in a platform-independent high level language and applies these policies by transforming program code. A policy generator takes resource descriptions, safety policies, platform interface and the application to be transformed and then generates a policy description file. This file is used by an application transformer to make the necessary changes to the application. The application transformer replaces system calls in the application to functions in a policy-enforcing library. Naccio has been implemented for both Win32 and JAVA platforms. Security Automata SFI Implementation (SASI) [23] uses a security automaton to specify security policies and extends the idea of software fault isolation by merging security policy into the application itself. The security automaton acts as a reference monitor for the code. In relation to a particular system the events that the reference monitor controls are represented by the alphabet, and the transition relationship encodes the security policy enforced by the reference monitor (so called IRM, Inline Reference Monitor [5]). The security automaton is merged into application code by an IRM binary rewriter. It adds code that implements the automaton directly before each instruction. The rewriter is language specific: x86 SASI is the implementation for x86 machine code and JVML SASI is for JAVA virtual machine.

PoET (Policy Enforcement Toolkit) [6] is an implementation of IRM for JAVA applications. Superior to Naccio and SASI, which requires the source code information or binaries generated with special compiler, PoET does not require any source code information available and the code after being rewritten can be executed on any version of JVM. It uses relatively higher level, event-oriented, JAVA-like language to specify security policies, which is called PSLang. Specifying a security policy involve defining [5]: *Security events* to be mediated by the reference monitor; *Security state that* stored about earlier security events and is used to determine which security events can be allowed to proceed; *Security updates that* update the security state, signal security violations and/or take other remedial action when the related security event happens. PoET adds the security enforcement according to the instructions specified in the security policy.

3. BRSS: Binary Rewriting Security System

We designed BRSS (Binary Rewriting Security System) with two goals: 1) reduce the administrative overhead of enforcing mobile code security and 2) reduce the performance overhead of adding binary rewriting to proxy servers. In this section we describe BRSS, focusing on its integration with web proxies.

3.1. Overview and Architecture

We integrated BRSS into caching proxy servers (as shown in Figure 1). A proxy server stays between clients and remote Internet servers and transparently mediates all requests for web objects. Proxy servers usually provide caching service, which improves access speed, reduces the load on networks and servers and increases availability by replicating information [37]. Sitting between the proxy and cache, BRSS secures the recipients by intercepting untrusted mobile code and rewriting it according to the specified security requirements. This provides system administrators an easy way to enforce security policies at the level of administrative domains without configuring each individual machine. BRSS was designed to utilize the existing caching facilities in the proxy server. When a client sends a applet request to the proxy and if the proxy serve can not find it in the cache (which is called a cache miss), the proxy forwards the request to the remote server and BRSS intercepts and rewrites the applet files returned from the remote server. Thus an overhead is added in the process. The rewritten and hence secured code is then cached in the proxy cache and distributed to the client for all subsequent requests. The whole process is transparent to the proxy cache and cache validation is handled by the cache as it used to be. Subsequent requests for the same applet may have already been rewritten and cached in the proxy cache (which is called a cache *hit*). In this case, the request can be satisfied directly by the rewritten code from the cache without the intervention of BRSS. With the cache size increases, more and more rewritten applet files are cached and the hit ratio increases, therefore the overhead added by BRSS will be amortized by the increasing number of applet requests satisfied by cached rewritten code.

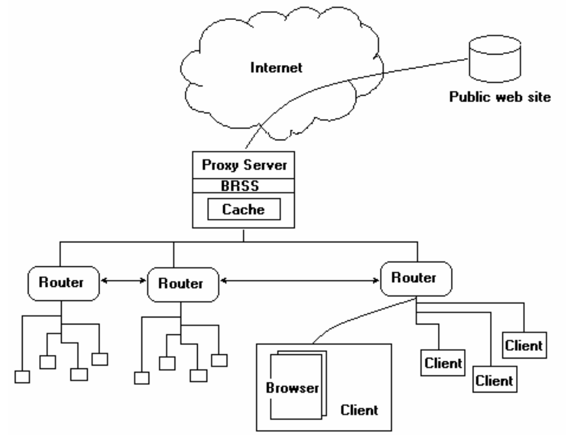


Figure 1. BRSS overview

As shown in Figure2, BRSS has four parts: Database, Application Management, Policy Editor, and Binary Rewriter. *Database* stores security policies. It organizes the security policies into several categories such as file system, network communication and memory usage and differentiates the policies within each category into different security levels. The database also keeps the history of changes made to binaries that are rewritten. *Application Management*, a graphical user interface, is provided for administers to manage the database and binary rewriter conveniently. The *Policy Editor* reduces the complexity of creating security policies by wrapping the JAVA Virtual Machine Library (JVML) with an abstract intermediate level. Detailed description of the management system can be found in our previous work [34]. The *Binary Rewriter* is actually a plug-in module for real binary rewriters. It provides two buffers for the binary rewriter. The real binary rewriter gets its input from the receiving buffer and rewrites the code to the output buffer. A *content filter* inserted at the proxy intercepts and filters out the mobile code that pass by to the binary rewriter's receiving buffer.

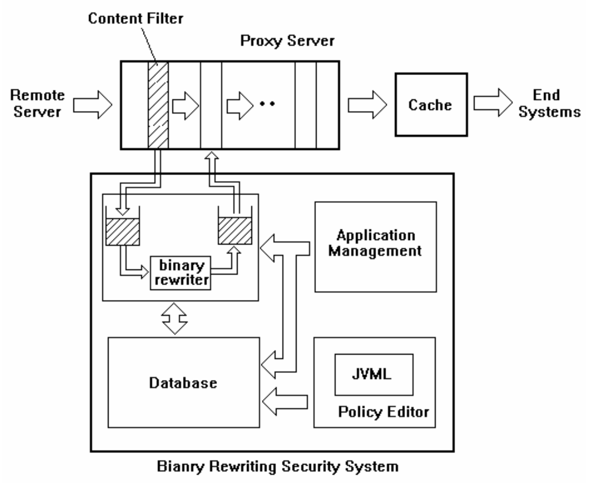


Figure 2. BRSS system architecture

3.2. Implementation Details

We chose PoET [6] as the binary rewriter for BRSS because it provides all the features mentioned in previous section and we have been using it in previous work[34]. As mentioned previously, PoET is a command-line JAVA application with high initialization overhead. To reduce the overhead, we modified PoET and allowed it to run in the background as a multithreaded server preloaded with a specified set of security policies. BRSS can also support other binary rewriters. The binary rewriter can be supplied as a plug-in module that is easily replaceable.

We selected Apache as the web proxy server. Apache is more well-known as web server rather than proxy server. The reason that we chose Apache is because it provides a filter mechanism so that we could easily add a content filter that integrates BRSS with the proxy server. Also, the performance of Apache as proxy server is good enough for our prototype system.

A skeleton of the content filter is shown in Figure 3. When the proxy gets a request, it first checks if the request can be served from its cache in the *cache_url_handler* function. If it is a cache hit, the *cache_out* filter is added to get the data from the cache, otherwise, it adds a *cache_in* filter before all the other output filters to save servers' reponse into the cache. We thus placed our security filter whenever a cache_in filter is needed and made it process the data flow before the cache_in filter. JAVA applet traffic can be identified by analyzing the URLs. Once it's found, the complete file will be pulled out by the *pull_data_out* function, the function *notify_PoET* will be called to send the notification to PoET, after which the applet file will be pushed back into the dataflow by *push_data_in* function.

```
/*seucrity filter*/
static int security_in_filter(..)
{
    //pull the ".class" file out
    rv = pull_data_out(…);
}
/*register the filter*/
static void register_hooks(…)
{
    security_in_filter_handle= ap_register_output_filter(..,
 security_in_filter, …);
    cache_in_filter_handle = ap_register_output_filter(…,
  cache_in_filter, …);
}
/*analyze the url*/
static int cache_url_handler(…)
{
    //if the request can be served from the cache
    ap_add_output_filter_handle(cache_out_filter_handle, …);
    //if it can not add the cache_in filter and add the security filter
         ap_add_output_filter_handle(security_in_filter_handle, ..);
         ap_add_output_filter_handle(cache_in_filter_handle, …);
}
/* pull out the class file */
static apr_status_t pull_data_out(… )
{
    //send notification to PoET
    if ( notify_PoET(…) > 0)
    {
           //push the rewritten code back to the data flow;
          if ((rv = push_data_in(…))!= APR_SUCCESS)
                        …
       }
}
```

Figure 3. The skeleton of the content filter added to Apache2.0

4. Performance Evaluation

In this section, we study the performance overhead of adding BRSS to an Apache proxy. We chose Web Polygraph [31] as our benchmarking tool because it is a freely available and widely accepted benchmark [40] that is specifically for proxy performance measurement. In order to simulate the real applet file traffic, we examined applet traffic characteristics by analyzing the raw proxy cache access logs. We analyzed the overhead added by BRSS on JAVA applets requests. We then compared the proxy's overall performance with and without BRSS under different request rates.

4. 1. Web Polygraph and the Workload Model

Web Polygraph is designed specially for caching proxy benchmarking. Its latest workload model, Polymix-4, includes many key web traffic characteristics, such as synthetic workload composed of various content types, specified request rates and inter-arrival times, a mixture of cache hits and cache misses, etc. Most importantly, Web Polygraph has the capability to simulate real content traffic, which is crucial for testing our binary rewriting system. Table 1 summarizes the content types that the Web Polygraph server uses for benchmarking. The server hosts mainly three content types (i.e. image, HTML, download.) with specific file extensions. All other content types are included in the 'other' type without specifying file extensions. In order to test our BRSS, we need the server to generate real applet file traffic. We will describe in detail the applet request traffic model in the next section. The size models of content types listed in Table 1 are generated by analyzing unique file size transportation from the web proxy log [32].Table 2 lists the default parameters of the polymix-4 workload model. If not specially stating, we use the polymix-4 workload model by default.

Table 1. Content types of Web Polygraph workload

Type	Percen tage	Reply Size Distribution	Cacha bility	Extensions
Image	65%	Exponential(4.5KB)	80.00%	.gif, .jpeg, and .png
HTML	15%	Exponential(8.5KB)	90.00%	.html and .htm
download	0.5%	Log-normal(300KB, 300KB)	95.00%	.exe, .zip, and .gz
Other	19.5%	Log-normal(25KB, 10KB)	72.00%	

Table 2 Default parameters of the workload model

Client request rate	0.4/secs
Number of transactions per connection	Zipf (64)
Request types	IMS: 20% Reload: 5% Basic 75%
Server delay	40 millisecond/packet
Server think time	Normal distribution (2.5, 1)

4.2. Applet File Traffic Model

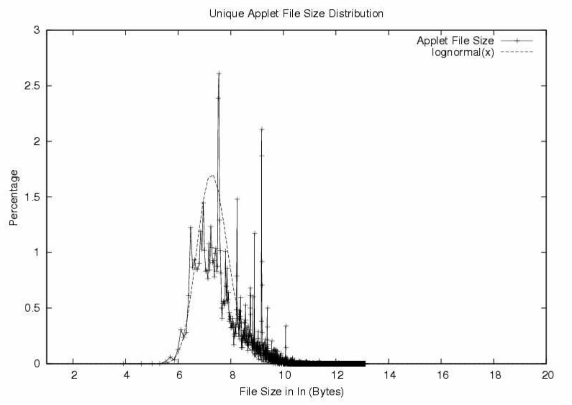
Many studies have been performed on analyzing Internet traffic characteristics[28,19,20]. However, to our knowledge, no specific analysis on JAVA applet file traffic model has ever been published in the literature. So we conducted our own research on applet file traffic. In this section, we present our applet traffic model by analyzing the raw proxy logs.

We used the raw access logs and daily summary reports provided by the IRCache project [33]. These logs were collected on a daily basis between July 11th, 2004 and August 11th, 2004 from a number of proxy servers located at various educational and commercial institutions throughout the United States [33]. The access log records the client's request information for HTTP object and consists mainly of the entries of time, duration, client address, result codes, bytes, request method, URL, type and etc. Therefore, applet file requests can be identified by their extension (i.e. .class) in the corresponding URL entries and from the 'result codes' entry we can determine the status of the specific transaction. Table 3 shows the statistic summary for the raw data set. The successful requests mean the transactions whose HTTP result code is "200/OK". From this table we can calculate that the applet file requests take 0.08% of the total requests. Among the applet file requests, 27.08% are successful requests. The main errors have the code 304 (not modified) 31.3%, 404 (not find) 27.1% and 503 (Service Unavailable) 32.6%. Among the successful request, 59.6% are unique files requests.

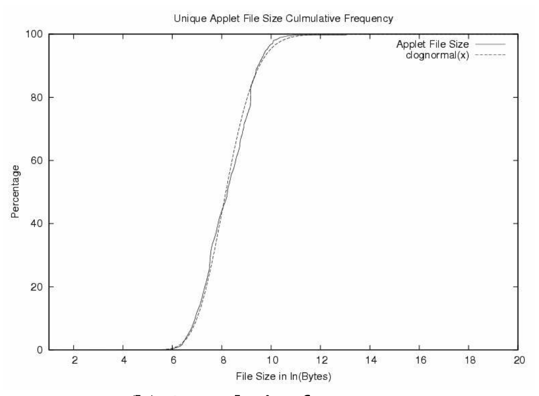
Table 3. Summary of access log characteristics

Access log Duration	08/11/ 2004 ~ 09/10/ 2004
Total requests count	117790462
Total applet request count	95054
Total successful applet requests count	25737
Total unique successful applet requests count	14646
Mean size of successful applet requests	7176.85
Median size of successful applet requests	3725
Mean size of successful unique applet file	7085.56
Median size of successful unique applet file	3775

Figure 4 shows the distribution of all successful unique applet files in the data set. Figure 4a compares the distribution with the synthetic lognormal distribution (the dashed line) with parameters μ= 7.61 and σ = 0.66. Figure 4b is the corresponding cumulative frequency plot. As we can see, the applet file size distribution is close to log normal distribution.



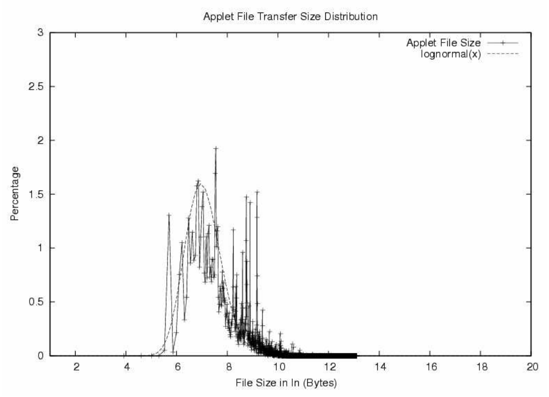
(a) Frequency



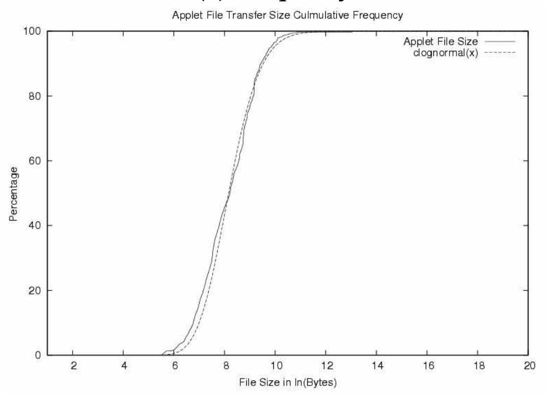
(b) Cumulative frequency

Figure 4. Unique applet file size distribution

Figure 5 shows the transfer size distribution of all successful applet file in the data set. Figure 5a shows the distribution is close to the lognormal distribution (the dashed line) with parameters $\mu = 7.06$ and $\sigma = 0.70$. Figure 5b is the corresponding cumulative frequency plot.



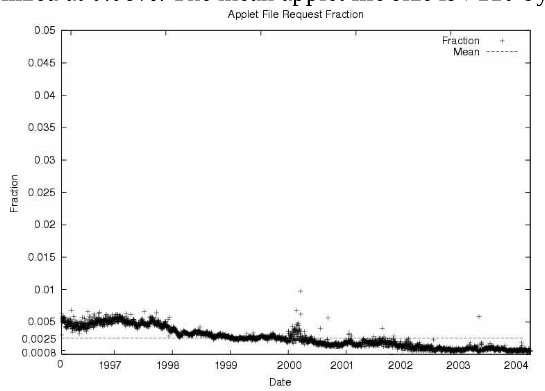
(a) Frequency



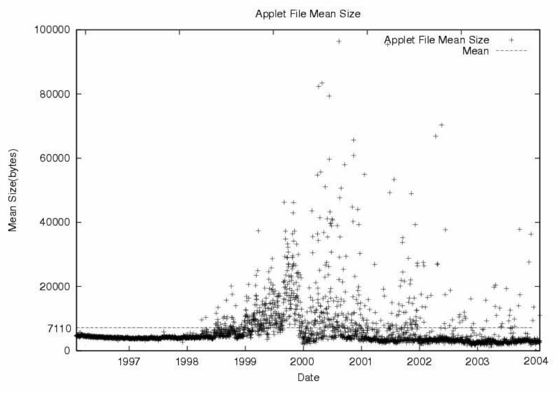
(b) Cumulative frequency

Figure 5. Successful applet file transfer size distribution

We also analyzed the IRCache daily summary report of applet files in an extended period from 1997 to 2004. Figure 6 shows the daily applet file request ratio (6a) and the daily mean size (6b) respectively. The dashed line is the mean value. From these figures we can see that the applet file requests rate started from 0.6% in the year of 1997, dropped after the year of 2002 and tends to be stabilized at 0.08%. The mean applet file size is 7110 bytes.



(a) Applet request ratio



(b) Mean size of applet requests

Figure6. Applet file request daily stat (1997~2004)

We decided to use the applet traffic model with the parameters shown in Table 4 in our experiment. We took 0.25 percentages (the mean percentage) from the 'other' type of Polymix-4 workload as the applet file request percentage. We built up the applet file generation database on the server side with real applet files downloaded randomly from the Internet and with the file sizes according to the unique applet file size distribution. All the other paraments such as cachability, object life cycle, etc, we use the same as the 'other' type.

Table 4. Applet traffic model

Percentage	0.25%
File size distribution	Lognormal(7.61, 0.66)
Other paraments	Use default as 'other' type

4. 3 Experiment Setup

We used one pair of client and server in our experiments. The hardware configuration is as following: the client runs on a PC with 756Hz AMD CPU and

256MB. The server and Apache proxy run on PCs with 2GHz Pentium4 CPU and 640 MB memory respectively. All the three machines are connected through 100M network. In addition, we installed name service and network time(ntp) service on the proxy server, which are required by the Web Polygraph benchmark. The proxy server software that we used is Apache version 2.0. At the time of our experiments, the garbage collection functionality hasn't been implemented on Apache 2.0 but the same functionality works properly on Apache 1.3. We thus rewrote the garbage collection part in version 1.3 and added it to Apache2.0.

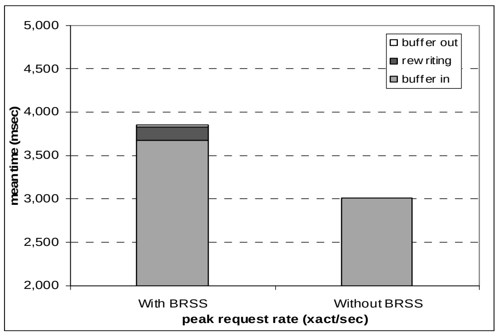
Web Polygraph uses synthetic clients and servers. Given a specified Peak Request Rate (briefly PRR), a number of clients will be created to generate the specified PRR and an according number of servers will also be created to response to the requests. Therefore, by varying the PRR, we are actually changing the number of clients. In our proxy configuration, we set the max client number that the Apache can sustain concurrently to be 250, which is approximately according to 30xacts/sec PRR. All the other Apache proxy paraments are by default.

4. 4 Experiment Result

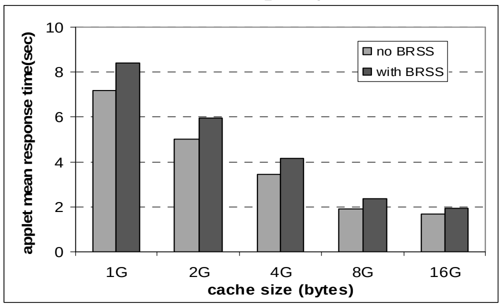
BRSS adds overhead to the applet file requests. In our experiments, we want to learn exactly where the overhead come from, how the cache size affects the overhead, and how the overhead affects the proxy's overall performance.

4.4.1 The Overhead of BRSS on applet file requests. To measure the overhead of BRSS adds specifically on applet file requests, we kept a log on the proxy server to record the applet cache miss time, in which case the requested applet file is not cached and the proxy forwards the client's applet file requests to the remote server until it gets the response data back and sends it to the client. With BRSS, this time can be further divided the time into: *buffer in*, which is the time from the proxy gets the client's request for applet file until the receiving buffer buffers the entire applet file; *rewriting*, which is the time that the binary rewriter takes to rewrite the file and *buffer out*, which is the time that the rewritten file in the output buffer is cached and sent to the client. Figure 7a shows the comparison result. We use the PRR 30 xacts/sec for the experiment, under which the proxy is fully loaded. As we can see that the overhead added to the applet request is mainly caused by *buffer in* time. This is because the binary rewriter requires the entire file to be cached before it can rewrite it while, instead, without BRSS the file will be cached and then sent to the client in the unit of a trunk of the file. The rewriting time adds relatively little to the overhead. As shown in our previous work [34][35], the rewriting time has also relationship with the binary file size, the complexity of the security policy and the number of places in the binary file that needs to add monitoring code in. In our experiments, we used the security policy limitMem, which is used to limit the amount of the memory that the application is allowed to use and has the worst performance overhead in our previous measurements [34]. The buffer out is merely the time required to transfer files from the proxy to the clients and thus is very small that can barely be seen from the figure.

To see how varying the cache size affects the overhead added by BRSS on the applet files, we logged each transaction for applet file quest on the client side. The time measured thus is the mean value of both cache miss and cache hit. Figure 7b compares the result with and without BRSS under different cache sizes. The PRR used is 30 xacts/sec. We can see that with the cache size increases and so the number of cached rewritten applet files and cache hits increases, the mean response time difference between with and without BRSS is decreasing, which means the overhead caused by BRSS as shown in Figure 7a is amortized by the increased cache hits.

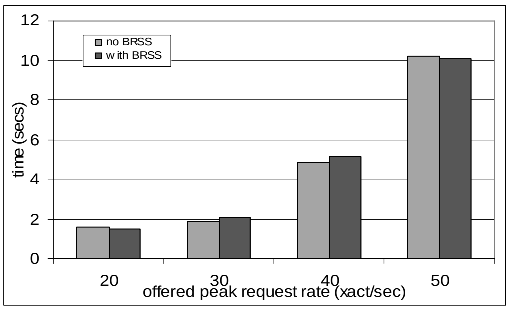


(a) Time measured at the proxy (with 16G cache)

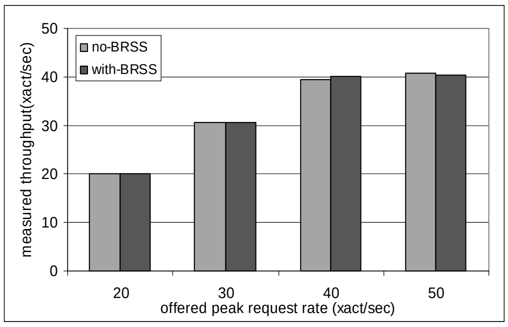


(b) Mean response time under different cache size
Figure 7 Overhead added by BRSS on applet file request (PRR 30 xacts/sec)

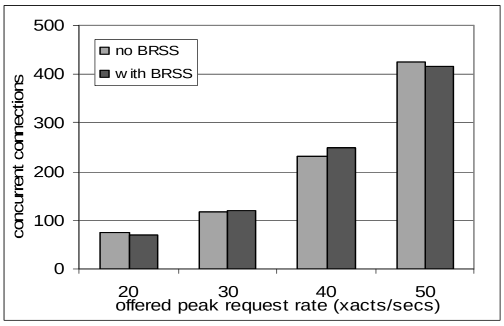
4.4.2 The Overall performance. To find out how BRSS affects the proxy's overall performance, we compared the proxy's performance with and without BRSS under different PRRs that the proxy server becomes from lightly loaded to badly overloaded. The result is shown in Figure 8. As the PRR increases, the mean response time and throughput increases. After the 30 xacts/sec, the proxy becomes overloaded: The response time increases dramatically; The concurrent level increases while the throughput tends to be stabilized at 40 xacts/sec, which means client requests keep queued at the proxy; The error rate rises up to be over 1% and over 90% of the errors are caused by connection time out; The CPU usage tends to be stable at 12%. We can see that except a little overhead on the server's CPU usage (which is less than 1%) can be perceived constantly, the performance difference between with and without BRSS is very small and within the measurement error range.



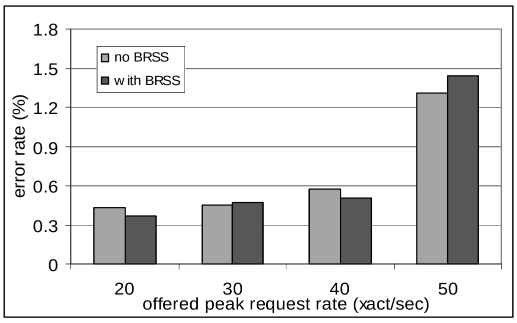
(a) Mean response time



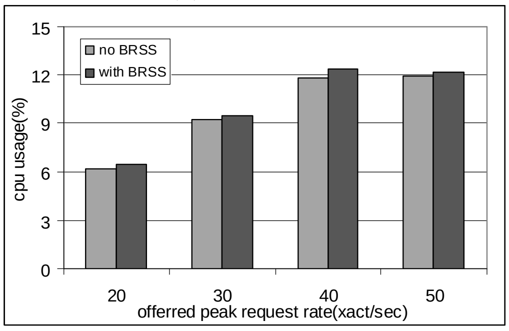
(b) Throughput



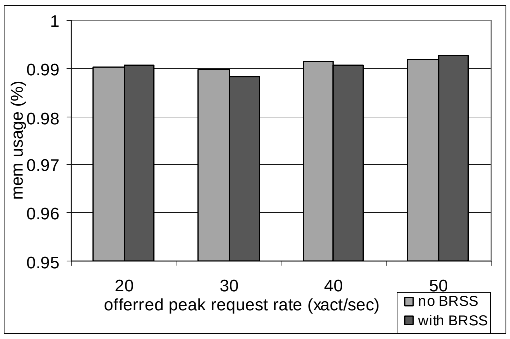
(c) Concurrent level



(d) Error ratio



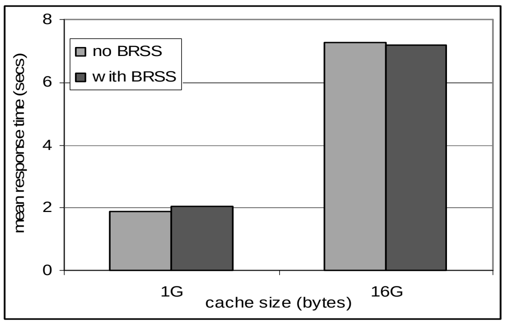
(e) CPU usage



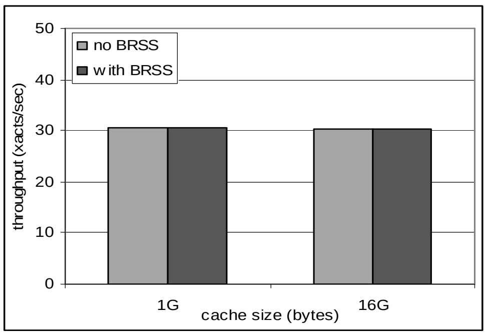
(f) Memory usage

Figure 8. Apache proxy performance under different request rates (16G Cache Size)

In the above experiment we set the cache size to 16G bytes. The reason is that under heavy load (e.g. 50 xacts/sec PRR) and at the same time with a small cache (e.g., 1G), the Apache proxy's behavior become very bad and underterministic. This is consistent with Cao's result [29]. However, to testify that our above result is also consistent with small cache, we give the overall performance under 30 xact/sec PRR with 1G cache size as shown in Figure 9. As we can see that with different size of cache, the overall performance with and without BRSS is still almost the same.



(a) Mean response time



(b) Throughput

Figure 9. Overall performance under 30 xacts/sec PRR

Therefore, to summarize it, BRSS almost has no impact on the proxy's overall performance under different workloads and cache sizes. The overhead added specially on JAVA applet requests can be amortized by increasing cache size.

5. Related Work

Applet Trap[27] is a commercially available anti-virus software. It wraps applets in security monitoring code before the gateway server passes the applets on to the requesting client computer. The applets run their original code along with the monitoring wrapper which looks ahead into the applets' behavior to determine if the action applets will take matches any behavior defined in the administrators' policy as malicious (such as reformatting the hard disk). Different from many other content filtering or blocking software, the execution of the monitoring code doesn't need host system or external software support. Although there is little material available about how AppletTrap rewrites the JAVA applet code, we suspect that AppletTrap can only add checking code to block the predefined malicious instructions while BRSS enforces more comprehensive security policies.

WiSA (Wisconsin Safety Analyzer) [26] is an on-going project that aims at developing analysis techniques especially suited for COTS components. The General Purpose Binary rewriter is one of its subtopics. Its goal is to provide a flexible and extendable general infrastructure that utilizes existing binary rewriting and analyzing tools such as EEL and codesurfer, and works across different platform, architectures and languages. Their work will be greatly beneficial to our prototype. However, to bring all of them into one data structure is a great challenge.

M. Arlitt and C. Williamson [28] did a comprehensive workload characterization study of a World-Wide Web proxy. We followed the similar method in our analysis of JAVA applet file traffic. WPB (Wisconsin Proxy Benchmark) [29] is another caching proxy benchmark tool that is similar to Web Polygraph. WPB also uses synthetic clients and server processes. The workload is generated by reproducing the workload characteristics found in web proxy traces. The main performance data collected by the benchmark are latency, proxy hit ratio and byte hit ratio, and number of client errors. However, it can not generate real content traffic as Web Polygraph does, which makes it impossible for content filtering types of performance evaluation.

6. Conclusion

In this paper, we presented BRSS, a binary rewriting security system for mobile code. BRSS integrates binary rewriting with proxy caching techniques. It provides efficient support and management for binary rewriters and mobile code. In addition, in this research we also developed a unique methodology for our performance evaluation section. We first performed a study specifically on JAVA applet file Internet traffic characteristics. This is the first time, to our knowledge, that this kind of study has been conducted while most previous work has focused on the overall Internet traffic. We employed the Web Polygraph as our performance benchmark, but customized it to fit our requirement for real content filtering type tests. We verified through comprehensive experiments that adding BRSS into the proxy server almost has no impact on the overall proxy performance and the overhead added by BRSS on applet files is amortized as more requests are satisfied by proxy cache.

Acknowledgements

We would like to thank Fred Schneider and Cornell University for allowing us to user PoET for this research. We would like to thank Alex Rousskov from the Measurement Factory for his generous help on our experimental setup. We would also like to thank Duane Wessels from IRCache for allowing us to access the daily raw access log.

References

[1] J. Feigenbaum, P. Lee, Trust Management and Proof-Carrying Code in Secure Mobile-Code Applications, *DARPA Workshop on Foundations for Secure Mobile Code* in Monterey, CA, March, 1997.

[2] Roshan Thomas, A Survey of Mobile Code Security Techniques, *22nd National Information Systems Security Conference*, Oct, 1999

[3] Greg Andrews, Link-Time Optimization of Parallel Scientific Programs, University of Arizona Seminar Abstract, Nov, 2002.

[4] S. Debray, W. Evans, R. Muth, B. D. Sutter, Compiler Techniques for Code Compaction, *ACM Transactions*

on Programming Languages and Systems. ACM Press. Vol. 22 (2). 2000. pp. 378-415

[5] F. B. Schneider, etc, A Language-Based Approach to Security, *Informatics*, 2001, pp. 86-101.

[6] U. Erlingsson, F. B. Schneider, IRM Enforcement of JAVA Stack Inspection, *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[7] M. Gasser, Building a Secure Computer System, *Van Nostrand Reinhold International Company Limited*, April, 1988, pp. 162-166, pp. 93-127.

[8] Secure Computing with JAVA: Now and the Future: http://java.sun.com/marketing/collateral/security.html.

[9] C. Cifuentes, M. Emmerik, UQBT: Adaptable Binary Translation at Low Cost, *Computer*, Vol 33, No 3, March 2000, IEEE Computer Society Press, pp 60-66

[10] T. Romer, et al., Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the First USENIX Windows NT Workshop*, Seattle, WA, August 1997.

[11] Byte Code Engineering Library (BCEL) http://jakarta. Apache.org/bcel /manual.html

[12] A. Srivastava and A. Eustace, ATOM: A system for building customized program analysis tools, *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, May 1994.

[13] L. R. James and E. Schnarr, EEL: Machine independent executable editing, *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[14] T. Chiueh and F. Hsu. Rad: A compile-time solution to buffer overflow attacks, *21st International Conference on Distributed Computing*, Phoenix, Arizona, April 2001, pp 409.

[15] M. Prasad and T. Chiueh, A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks, *Proceedings of Usenix Annual Technical Conference*, San Antonio, TX, June 2003

[16] B. Buck and J. K. Hllingsworth, An API for runtime code patching, *the International Journal of High Performance Computing Applications*, vol. 14, no. 4, 2000, pp. 317-329.

[17] Libsafe library: http://www.bell-labs.com/org/11356/ libsafe.html.

[18] C. Cowan et al., Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998, pp 63-78.

[19] M. Arlitt and C. Williamson, Internet Web Servers: Workload Characterization and Performance Implications, *IEEE/ACM Transactions on Networking* , Vol. 5, No. 5, October 1997, pp. 631-645.

[20] A. Mahanti and C. Williamson, Web Proxy Workload Characterization, Progress Report, Computer Sciences Dept, Univ. of Saskatchewan, Feb. 1999

[21] D. C. DuVarney, V.N. Venkatakrishnan and S. Bhatkar, SELF: a Transparent Security Extension for ELF Binaries, *New Security Paradigms Workshop*, Ascona, Switzerland, August 2003.

[22] D. Evans and A. Twyman, Flexible Policy-Directed Code Safety, *1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9-12, 1999.

[23] Ú. Erlingsson and F. B. Schneider, SASI enforcement of security policies: A retrospective, *Proceedings of the New Security Paradigms Workshop,* Ontario, Canada, September 1999, pp87-95.

[24] R. Wahbe, et al., Efficient Software-Based Fault Isolation, *Proceedings of the Symposium on Operation System Principles*, 1993.

[25] R. Hastings and B. Joyce, Purify: A tool for detecting memory leaks and access errors in C and C++ programs, *Proceedings of the Winter 1992 USENIX Conference*, Berkeley, CA, January 1992, pp 125–138.

[26] Wisconsin Safety Analyzer (WiSA): www.cs.wisc. edu/wisa/

[27] AppletTrap: www.trendmicro.com.au/product/ isap/

[28] M. Arlitt, etc., Workload Characterization of a Web Proxy in a Cable Modem Environment, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27, No. 2, September 1999, pp. 25-36.

[29] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark, Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998.

[30] Squid Document. http://www.squid-cache.org/

[31] Web Polygraph: http://www.web-Polygraph. org

[32] The Mesurement Factory Document about server workload file size. http://www.measurement-factory.com/docs/FAQ/pmix4-reply-size-distr/

[33] IRCache Project: http://www.ircache.net/

[34] Y. Song and B. D. Fleisch, Rico: A Security Proxy for Mobile Code, *Journal of Computers and Security Elsevier Advanced Technology*, Elsevier Press, Volume 23, Issue 4, 2004, pp. 338-351

[35] Y. Song and B. D. Fleisch, Sandboxing Mobile Code from Outside the OS, *in the 19th ACM Symposium on Operating Systems Principles,* Work in Progress Session, Bolton Landing, New York, 2003.

[37] Apache Web Proxy: http://httpd.Apache.org/docs-2.0/

[40] The Fourth TMF Cache-Off: http://cacheoff.measure ment -factory.com/

List of Written Publications (UC Irvine)

Peer-Reviewed Book Chapters

- M. Franz; “A Fresh Look At Low-Power Mobile Computing”; in L. Benini, M. Kandemir, J. Ramanujam (Eds.), *Compilers and Operating Systems for Low Power*; Kluwer Academic Publishers, Boston, ISBN 1-4020-7573-1, pp. 209-220; September 2003.

- M. Franz; “Safe Code—It’s Not Just For Applets Anymore”; in L. Boeszoermenyi and P. Schojer (Eds.), *Modular Programming Languages: Proceedings of the Sixth Joint Modular Languages Conference (JMLC 2003)*, Klagenfurt, Austria; Springer Lecture Notes in Computer Science, No. 2789, ISBN 3-540-40796-0; pp. 12-22; August 2003. (Full Text of Invited Keynote Address)

Peer-Reviewed Journal Papers

- M. Franz, D. Chandra, A. Gal, V. Haldar, Ch. W. Probst, F. Reig, and N. Wang; “A Portable Virtual Machine Target For Proof-Carrying Code”; *Science of Computer Programming*, Special Issue on Interpreters, Virtual Machines, and Emulators; accepted for publication.

Rigorously Peer-Reviewed Conference Papers

- A. Gal, Ch. W. Probst, and M. Franz; “Structural Encoding of Static Single Assignment Form”; to appear in *4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV’05)*, Edinburgh, Scotland; April 2005.

- A. Gal, Ch. W. Probst, and M. Franz; “Integrated Java Bytecode Verification”; in *1st International Workshop on Abstract Interpretation of Object-Oriented Programming Languages (AIOOL’05)*, Paris, France; January 2005.

- J. von Ronne, N. Wang, and M. Franz; “A Virtual Machine for Interpreting Programs in Static Single Assignment Form”; in *Proceedings of the ACM SIGPLAN 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME’04)*, Washington, D.C., pp. 23-30; June 2004.

- M. Beers, Ch. Stork, and M. Franz; "Efficiently Verifiable Escape Analysis"; in M. Odersky (Ed.), *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, Springer Lecture Notes in Computer Science, Vol. 3086, ISBN 3-540-22159-X, pp. 75-95; June 2004.

- V. Haldar, D. Chandra, and M. Franz; "Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing"; in *Proceedings of the 3rd USENIX Virtual Machine Research & Technology Symposium (VM'04)*, San Jose, California, ISBN 1-931971-20-X, pp. 29-41; May 2004. (Best Paper Award)

- Ch. W. Probst, A. Gal, and M. Franz; "Code Generating Routers: A Network-Centric Approach to Mobile Code"; in *Proceedings of the 2003 IEEE 18th Annual Workshop on Computer Communications (CCW'2003)*, Dana Point, California, IEEE Press, ISBN 0-7803-8239-0, pp. 179-186; October 2003.

- M. Franz, D. Chandra, A. Gal, V. Haldar, F. Reig, and N. Wang; "A Portable Virtual Machine Target For Proof-Carrying Code"; in *Proceedings of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, San Diego, California, pp. 24-31; June 2003.

- V. Haldar, Ch. Stork, and M. Franz; "The Source Is The Proof"; in C. Serban, S. Saydjari (Eds.), in *Proceedings of the ACM SIGSAC 2002 Workshop on New Security Paradigms (NSPW-2002)*, Virginia Beach, Virginia, ACM Press, ISBN 1-58113-598-X, pp. 69-73; September 2002.

- D. Chandra, Ch. Fensch, W.-K. Hong, L. Wang, E. Yardimci, and M. Franz; "Code Generation at the Proxy: An Infrastructure-Based Approach to Ubiquitous Mobile Code"; in *Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS 2002)*, Malaga, Spain, June 2002.

- M. Franz; "A Fresh Look At Low-Power Mobile Computing"; in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001 (COLP'01)*, Barcelona, Spain, pp. 15.1-15.6; September 2001.

Further Conferences, Workshops, and Technical Reports

- A. Gal, Ch. W. Probst, and M. Franz; *SSA-Based Java Bytecode Verification*; Technical Report No. 04-22, School of Information and Computer Science, University of California, Irvine; October 2004.

- A. Gal, Ch. W. Probst, and M. Franz; *Proofing: Efficient SSA-based Java Verification*; Technical Report No. 04-10, School of Information and Computer Science, University of California, Irvine; April 2004.
- A. Gal, Ch. W. Probst, and M. Franz; *Complexity-Based Denial of Service Attacks on Mobile-Code Systems*; Technical Report No. 04-09, School of Information and Computer Science, University of California, Irvine; April 2004.
- A. Gal, Ch. W. Probst, and M. Franz; *Static Closure of Java Dynamic Class Loading*; Technical Report No. 03-32, School of Information and Computer Science, University of California, Irvine; September 2003.
- N. Wang and M. Franz; *A Practical Mobile-Code Format With Linear Verification Effort*; Technical Report No. 03-26, School of Information and Computer Science, University of California, Irvine; November 2003.
- N. Wang, P. S. Housel, G. Zhang, and M. Franz; *An Efficient XML Schema Typing System*; Technical Report No. 03-25, School of Information and Computer Science, University of California, Irvine; November 2003.
- A. Gal, Ch. W. Probst, and M. Franz; *Proofing: An Efficient and Safe Alternative to Mobile-Code Verification*; Technical Report No. 03-24, School of Information and Computer Science, University of California, Irvine; November 2003.
- A. Gal, Ch. W. Probst, and M. Franz; *A Denial of Service Attack on the Java Bytecode Verifier*; Technical Report No. 03-23, School of Information and Computer Science, University of California, Irvine; October 2003.
- V. Haldar, Ch. W. Probst, V. Venkatachalam, and M. Franz; *Virtual Machine Driven Dynamic Voltage Scaling*; Technical Report No. 03-21, School of Information and Computer Science, University of California, Irvine; October 2003.
- R. E. Diaconescu, L. Wang, and M. Franz; *Automatic Distribution of Java Byte-Code Based on Dependence Analysis*; Technical Report No. 03-18, School of Information and Computer Science, University of California, Irvine; October 2003.
- V. Venkatachalam, L. Wang, A. Gal, Ch. Probst, and M. Franz; *ProxyVM: A Network-based Compilation Scheme for Resource-Constrained Devices*; Technical Report No. 03-13, School of Information and Computer Science, University of California, Irvine; March 2003.

- N. Wang, M. Franz, and N. Dalton; *Enabling Efficient Program Analysis for Dynamic Optimization of a Family of Safe Mobile Code Formats*; Technical Report No. 02-24, Department of Information and Computer Science, University of California, Irvine; September 2002.

Professional Personnel Associated With The Project (UC Irvine)

Faculty

- Michael Franz

Post-Doctoral Researchers

- Roxana Diaconescu
- Won-Kee Hong
- Christian W. Probst
- Fermin Reig

Graduate Students

- Andreas Gal
- Matthew Beers
- Deepak Chandra
- Niall Dalton
- Christian Fensch
- Cristian Petrescu Prahova
- Vivek Haldar
- Songmei Han
- Jeffrey von Ronne
- Christian Stork
- Lei Wang
- Ning Wang
- Efe Yardimci

Visiting Researchers

- Bryan Fulton

Presentations at Meetings, Conferences, Seminars, etc. (UC Irvine)

- A. Gal; 1st International Workshop on Abstract Interpretation of Object-Oriented Programming Languages (AIOOL'05), Paris, France; January 2005.
- J. von Ronne; ACM SIGPLAN 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME'04), Washington, D.C.; June 2004.
- Ch. Stork; 18th European Conference on Object-Oriented Programming (ECOOP 2004), Oslo, Norway; June 2004.
- M. Franz; Sun Microsystems, Inc., Mountain View, California; May 2004.
- M. Franz; Southern California Parallel Processing and Computer Architecture Workshop, Los Angeles, California; May 2004.
- V. Venkatachalam; Southern California Parallel Processing and Computer Architecture Workshop, Los Angeles, California; May 2004.
- R. Diaconescu; Southern California Parallel Processing and Computer Architecture Workshop, Los Angeles, California; May 2004.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program Review Meeting, Annapolis, Maryland; May 2004.
- Ch. W. Probst; ONR Critical Infrastructure Protection, Mobile Code Program Review Meeting, Annapolis, Maryland; May 2004.
- V. Venkatachalam; ONR Critical Infrastructure Protection, Mobile Code Program Review Meeting, Annapolis, Maryland; May 2004.
- V. Haldar; 3rd USENIX Virtual Machine Research & Technology Symposium (VM'04), San Jose, California; May 2004.
- M. Franz; IFIP WG2.4 Working Meeting, Brisbane, Australia; March 2004.
- M. Franz; Google, Inc., Mountan View, California; December 2003.
- M. Franz; Transmeta, Inc., Santa Clara, California; December 2003.
- Ch. W. Probst; 2003 IEEE 18th Annual Workshop on Computer Communications (CCW'2003), Dana Point, California; October 2003.

- M. Franz; IFIP WG2.4 Working Meeting, Santa Cruz, California, August 2003.
- M. Franz; “Safe Code: It’s Not Just For Applets Anymore” (invited talk); Sixth Joint Modular Languages Conference (JMLC 2003), Klagenfurt, Austria; August 2003.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, Ithaca, New York; July 2003.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, Arlington, Virginia; June 2003.
- D. Chandra; ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME’03), San Diego, California; June 2003.
- M. Franz; 20th Gesellschaft fr Informatik (GI) Workshop on Programming Languages and Computing Concepts, Bad Honnef, Germany; May 2003.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, Irvine, California; January 2003.
- Ch. Fensch; 10th Workshop on Compilers for Parallel Computers (CPC 2003), Amsterdam, Netherlands; December 2002.
- E. Yardimci; 10th Workshop on Compilers for Parallel Computers (CPC 2003), Amsterdam, Netherlands; December 2002.
- M. Franz; IFIP WG2.4 Working Meeting, Dagstuhl, Germany; November 2002.
- V. Haldar; ACM SIGSAC 2002 Workshop on New Security Paradigms (NSPW-2002), Virginia Beach, Virginia; September 2002.
- D. Chandra; Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS 2002), Malaga, Spain; June 2002.
- M. Franz; 2nd Workshop on Intermediate Representation Engineering for Virtual Machines (IRE 2002), Dublin, Ireland; June 2002.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, State College, Pennsylvania; July 2002.
- M. Franz; 19th Gesellschaft fr Informatik (GI) Workshop on Programming Languages and Computing Concepts, Bad Honnef, Germany; May 2002.

- M. Franz; IFIP WG2.4 Working Meeting, Simon's Town, South Africa; March 2002.
- M. Franz; Southern California Parallel Processing and Computer Architecture Workshop, Irvine, California; February 2002.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, Melbourne, Florida; January 2002.
- M. Franz; Workshop on Compilers and Operating Systems for Low Power 2001 (COLP'01), Barcelona, Spain; September 2001.
- M. Franz; ONR Critical Infrastructure Protection, Mobile Code Program PI Meeting, Arlington, Virginia; July 2001.

References Cited

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Values in Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–11, San Diego, California, January 1988.

[2] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.

[3] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P Anderson Co., Fort Washington, PA, Apr. 1980.

[4] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr. 1998.

[5] A. W. Appel and K. J. Supowit. Generalization of the sethi-ullman algorithm for register allocation. *Software - Practice and Experience*, 17(6):417–421, 1987.

[6] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 333–346, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[7] D. F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. Technical Report CSD-98-1017, University of California, Berkeley, Oct. 5, 1998.

[8] D. Beuche, L. Büttner, D. Mahrenholz, W. Schröder-Preikschat, and F. Schön. JPure - Purified Java Execution Environment for Controller Networks. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES'2000)*. Kluwer Academic Press, Oct. 2001.

[9] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[10] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, May 2000.

[11] B. Bush, D. Simon, and A. Taivalsaari. The Spotless System: Implementing a Java System for the Palm Connected Organizer. Technical Report TR-99-73, Sun Microsystems, Feb. 1999.

[12] CERT Coordination Center, Carnegie Mellon University, http://www.cert.org.

[13] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction (CC)*, pages 98–105, Boston, MA, June 1982.

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, Martin, E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, 1981.

[15] R. M. Cohen. The defensive Java Virtual Machine specification version 0.5. Technical report, Computational Logic, Inc., May 1997.

[16] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.

[17] M. Cooperation. Microsoft .NET. `http://www.microsoft.com/net/`, 2003.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[19] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices*, 33(10):21–35, Oct. 1998.

[20] D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[21] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. In D. E. Denning and P. J. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press / Addison-Wesley, New York, 1998.

[22] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion detection systems. *Computer Networks*, 31(8):805–822, Apr. 1999. Special issue on Computer Network Security.

[23] B. Delsart, V. Joloboff, and E. Paire. JCOD: A Lightweight Modular Compilation Technology for Embedded Java. *Lecture Notes in Computer Science*, 2491, 2002.

[24] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, Mar. 2000.

[25] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM SIGPLAN Notices*, 37(1):307–318, Jan. 2002.

[26] M. Franz. A Fresh Look At Low-Power Mobile Computing. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power COLP'01*, Sept. 2001.

[27] S. Freund and J. C. Mitchell. A formal specfication of the java bytecode language and bytecode verifier. In L. Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 147–166, N. Y., 1–5 1999. ACM Press.

[28] S. N. Freund. The costs and benefits of java bytecode subroutines. In *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA*, oct 1998.

[29] S. N. Freund and J. C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[30] A. Gal, C. W. Probst, and M. Franz. An Applet performing a complexty-based Denial-of-Service attack on the verifier. Available at `http://nil.ics.uci.edu/exploit`.

[31] A. Gal, C. W. Probst, and M. Franz. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.

[32] A. Gal, C. W. Probst, and M. Franz. Proofing: An Efficient and Safe Alternative to Mobile-Code Verification. Technical Report 03-24, University of

California, Irvine, School of Information and Computer Science, November 2003.

[33] A. Gal, C. W. Probst, and M. Franz. Proofing: Efficient SSA-based Java Verification. Technical Report 04-10, University of California, Irvine, School of Information and Computer Science, April 2004.

[34] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[35] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[36] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 154–165, Los Alamitos, CA, May 11–14 2003. IEEE Computer Society.

[37] C. Hawblitzel and T. von Eicken. Luna: A flexible java protection system. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 391–403, New York, Dec. 9–11 2002. ACM Press.

[38] J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[39] D. Hopwood. Java Security Bug (Applets Can Load Native Methods). *RISKS Forum*, 17(83), 1996.

[40] Insignia Solutions. Jeode Platform: Java for Resource-constrained Devices, 2000. http://www.javaworld.com/javaworld/javaone00/j1-00-insignia.html.

[41] The Java Hotspot Virtual Machine, Technical White Paper, Sun Microsystems, Inc. 2001.

[42] JavaME Test Suite - Performance Discovery. http://www.dogada.com/javame.

[43] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, Mar. 1995.

[44] C. League, V. Trifonov, and Z. Shao. Functional Java Bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001.

Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[45] S. Lelait, G. R. Gao, and C. Eisenbeis. A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops. In K. Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*, volume 1383, pages 204–218, Lisbon, Portugal, March 28 - April 4 1998. Springer.

[46] X. Leroy. Bytecode verification on java smart cards. *Software Practice and Experience*, 32(4):319–340, 2002.

[47] X. Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[48] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[49] G. McGraw and E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, New York, NY, USA; London, UK; Sydney, Australia, 1998.

[50] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime. `http://research.microsoft.com/˜emeijer/papers/clr.pdf`, 2001.

[51] Microsoft Corporation. Microsoft Security Program: Microsoft Security Bulletin (MS99-045): Patch Available ”Virtual Machine Verifier” Vulnerability, 1999.

[52] S. Microsystems. *KVM - Kilobyte Virtual Machine White Paper.* *`http://java.sun.com/products/kvm/wp/`*. Palo Alto, CA, USA, 1999.

[53] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.

[54] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, January 1997.

[55] R. O'Callahan. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 70–78, San Antonio, Texas, 1999.

[56] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[57] Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[58] Z. Qian. Standard Fixpoint Iteration for Java Bytecode Verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[59] Redhat. Vulnerability in zlib library, Advisory ID: RHSA-2002:026-35, 2002.

[60] E. Rose and K. H. Rose. Lightweight Bytecode Verification. In *OOPSLA-Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.

[61] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbering and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 12–17, San Diego, California, January 1988.

[62] K. Sohr. Nicht verifizierter Code: eine Sicherheitslücke in Java. *JIT 1999*, 1999.

[63] K. Sohr. *Die Sicherheitsaspekte von mobilem Code*. PhD thesis, Universität Marburg, 2001.

[64] S. Soman, C. Krintz, and G. Vigna. Detecting malicious java code using virtual machine auditing. In *Proceedings of the 11th USENIX Security Symposium*, pages 153–168. USENIX, Aug. 2003.

[65] SPEC. JVM98 Benchmarks. `http://www.spec.org/jvm98`, 2001.

[66] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 62–73, San Francisco, California, 1995.

[67] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[68] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[69] Sun Microsystems. J2SE: Java 2 Platform, Standard Edition (J2SE). http://java.sun.com/j2se/.

[70] Sun Microsystems. JSR-000139 Connected Limited Device Configuration 1.1. `http://www.jcp.org/en/jsr/detail?id=139`.

[71] Sun Microsystems. CDC: An Application Framework for Personal Mobile Devices, http://java.sun.com/products/cdc/, 2003.

[72] Sun Microsystems Inc. Connected, Limited Device Configuration, and the K Virtual Machine, April 2000.

[73] Sun Microsystems, Inc. Java Card 2.2.1 Specification, Public Review Draft, 2003. http://java.sun.com/products/javacard/JavaCard221.html.

[74] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[75] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, Oct. 2000.

[76] V. Venkatachalam, L. Wang, A. Gal, C. W. Probst, and M. Franz. ProxyVM: A Network-based Compilation Infrastructure for Resource-Constrained Devices. Technical Report 03-13, University of California, Irvine, School of Information and Computer Science, 2003.

[77] D. J. Wetherall. *Service introduction in an active network*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998.

[78] F. Yellin. Low level security in Java. In O'Reilly and Associates and Web Consortium (W3C), editors, *World Wide Web Journal: The Fourth International WWW Conference Proceedings*, pages 369–380. O'Reilly & Associates, Inc., 1995.